CHAPTER 3 Programming for Performance

3.1 Introduction

The goal of using multiprocessors is to obtain high performance. Having understood concretely 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 understand 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 runtime 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 just for parallel software designers but also for architects. Besides helping us understand parallel programs as workloads for the systems we build, it also helps us appreciate hardware-software tradeoffs; that is, in what aspects of programmability and performance can the architecture be of assistance, and what aspects are best left to software. The interdependencies of program and system are more fluid, more complex and have far greater performance impact in multiprocessors than in uniprocessors; hence, this understanding is very important to our goal of designing high-performance systems that reduce cost and programming effort. We will carry it forward with us throughout the book, starting with concrete guidelines for workload-driven architectural evaluation in the next chapter.

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. 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 the issues and on software techniques. Architectural techniques, sometimes hinted at here, are the subject of the rest of the book.

While there are several interacting performance issues to contend with, they are not all dealt with 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 largely independent of the underlying architecture or communication abstraction, 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 interprocess communication, 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, one that 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 of the work among processes in the parallel program. The actual communication that occurs therefore depends not only on the partitioning but also 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. The relationship between communication, data locality and the extended memory hierarchy is discussed in Section 3.3. Then, Section 3.4 examines the software techniques to address the major performance issues in orchestration and mapping: techniques for reducing the extra communication by exploiting data locality in the extended memory hierarchy, and techniques for 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. While there are interactions and tradeoffs among all the performance issues we discuss, the chapter discusses each independently as far as possible and identifies tradeoffs as they are encountered. Examples are drawn from the four case study applications throughout, and the impact of some individual programming techniques illustrated through measurements on a particular cache-coherent machine with physically distributed memory, the Silicon Graphics Origin2000, which is described in detail in Chapter 8.The equation solver kernel is aso carried through the discussion, and the performance techniques are applied to it as relevant, so by the end of the discussion we will have created a high-performance parallel version of the solver.

As we go through the discussion of performance issues, we will develop simple analytical models 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; i.e. 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 factors 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. Together, they will help us learn how to use programs to evaluate architectural tradeoffs in the next chapter. This view and mapping are discussed in Section 3.5.

Once we have understood the performance issues and techniques, we will be ready to do what we wanted to all along: understand how to create high-performance parallel versions of complex, realistic applications; namely, the four case studies. Section 3.6 applies the parallelization process and the performance techniques to each case study in turn, illustrating how the techniques are employed together and the range of resulting execution characteristics presented to an architecture, reflected in varying profiles of execution time. We will also finally be ready to fully understand the implications of realistic applications and programming techniques for tradeoffs between the two major lower-level programming models: a shared address space and explicit message passing. The tradeoffs of interest are both in ease of programming and in performance, and will be discussed in Section 3.7. Let us begin with the algorithmic performance issues in the decomposition and assignment steps.

3.2 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 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*, and
- 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 fit 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 can be improved by spending more time determining a good assignment (extra work). The goal of decomposition and assignment is to achieve a good compromise between these conflicting demands. Fortunately, success is often not so difficult in practice, as we shall see. Throughout our discussion of performance issues, the four case studies from Chapter 2 will be used to illustrate issues and techniques. They will later be presented as complete case studies addressing all the performance issues in Section 3.6. For each issue, we will also see how it is applied to the simple equation solver kernel from Chapter 2, resulting at the end in a high performance version.

3.2.1 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 earlier—with proper assignment and reduced serialization, and 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 just how many calculations are done but 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, but they should 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 due to either mutual exclusion (waiting to enter critical sections) or dependences. The assignment step should ensure that reduced serialization is possible, and orchestration should ensure that it happens.

There are four parts to balancing the workload and reducing synchronization wait time:

- Identifying enough concurrency in decomposition, and overcoming Amdahl’s Law.
- Deciding how to manage the concurrency (statically or dynamically).
- Determining the granularity at which to exploit the concurrency.
- 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 the equation solver that concurrency may be found by examining the loops of a program or by looking more deeply at the fundamental dependences. Parallelizing loops usually leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure(s). 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 are 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 grid or array level 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 sub-operations or stages of the pipeline are performed concurrently. In encoding a sequence of video frames, each block of each frame passes through several stages: pre-filtering, convolution from the time to the frequency domain, quantization, entropy coding, etc. There is pipeline parallelism across these stages (for example a few processes could be assigned to each stage and operate concurrently), as well as 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). Orthogonal levels of 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 segments within a wire, and across the many routes evaluated for each segment (see Figure 3-1).

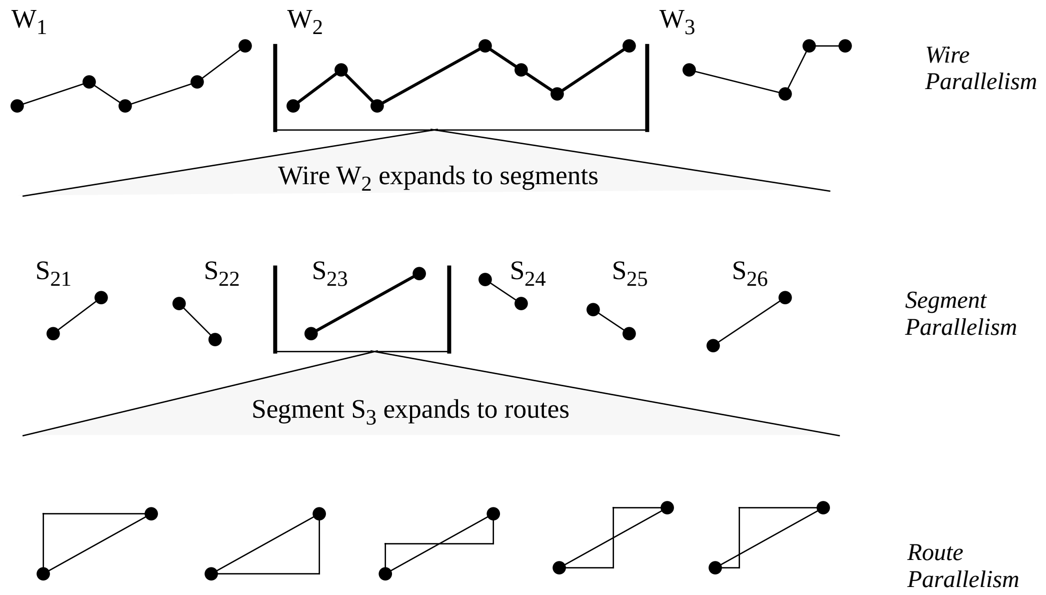


Figure 3-1 The three axes of parallelism in a VLSI wire routing application.

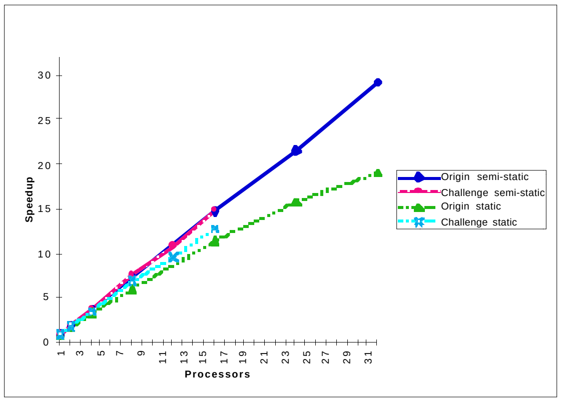
The degree of available function parallelism is usually modest and does not grow much with the size of the problem being solved. The degree of data parallelism, on the other hand, usually grows with data set size. Function parallelism is also often more difficult to exploit in a load balanced way, since different functions involve different amounts of work. Most parallel programs that run on large-scale machines are data parallel according to our loose definition of the term, and function parallelism is used mainly to reduce the amount of global synchronization required between data parallel computations (as illustrated in Ocean, in Section 3.6.1).

By identifying the different types of concurrency in an application we often find much more than we need. 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 we 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 the previous chapter, 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 the assignment does not adapt at runtime to other environmental considerations. Since the assignment is predetermined, static techniques do not incur much task management overhead at runtime. However, to achieve good load balance they require that the work in each task be predictable enough, or that there be so many tasks that the statistics of large numbers ensure a balanced distribution, with pseudo-random assignment. 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 limiting the robustness of static load balancing.

Dynamic techniques adapt to load imbalances at runtime. They come in two forms. In *semi-static* 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 runtime. That is, we profile (measure) the work that each task does in one phase, and use that as an estimate of the work associated with it in the next phase. This repartitioning technique is used to assign stars to processes in Barnes-Hut (Section 3.6.2), by recomputing 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 semi-static partitioning



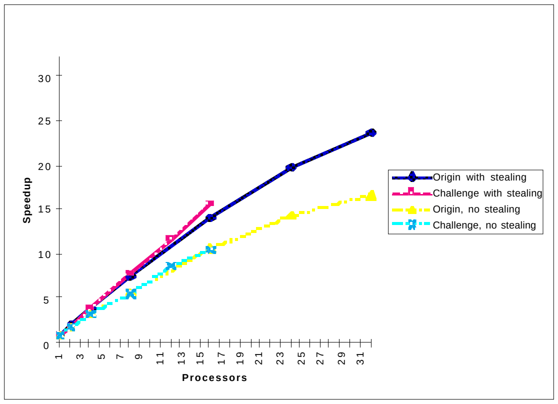


Figure 3-2 Illustration of the performance impact of dynamic partitioning for load balance.

The graph on the left shows the speedups of the Barnes-Hut application with and without semi-static partitioning, and the graph on the right shows the speedups of Raytrace with and without task stealing. Even in these applications that have a lot of parallelism, dynamic partitioning is important for improving load balance over static partitioning.

over a static assignment of particles to processors, for a 512K particle execution measured on the Origin2000. It is clear that the performance difference grows with the number of processors used.

The second dynamic technique, *dynamic tasking*, is used to handle the cases where either the work distribution or the system environment is too unpredictable even to periodically recompute a load balanced assignment.[1] For example, in Raytrace the work associated with each ray is impossible to predict, and 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. Each process repeatedly takes a task from the pool and executes it—possibly inserting new tasks into the pool—until there are no tasks 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 Raytrace application, for the balls data set 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), execute the iteration, and access the counter again, until no iterations remain. The task size can be increased by taking multiple iterations at a time, i.e., adding a larger value to the shared loop counter. In *guided self-scheduling* [AiN88] 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 there are few queue accesses 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 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 there are no more tasks 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,

1. The applicability of static or semi-static 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 may also not be appropriate for time-shared or heterogeneous systems.

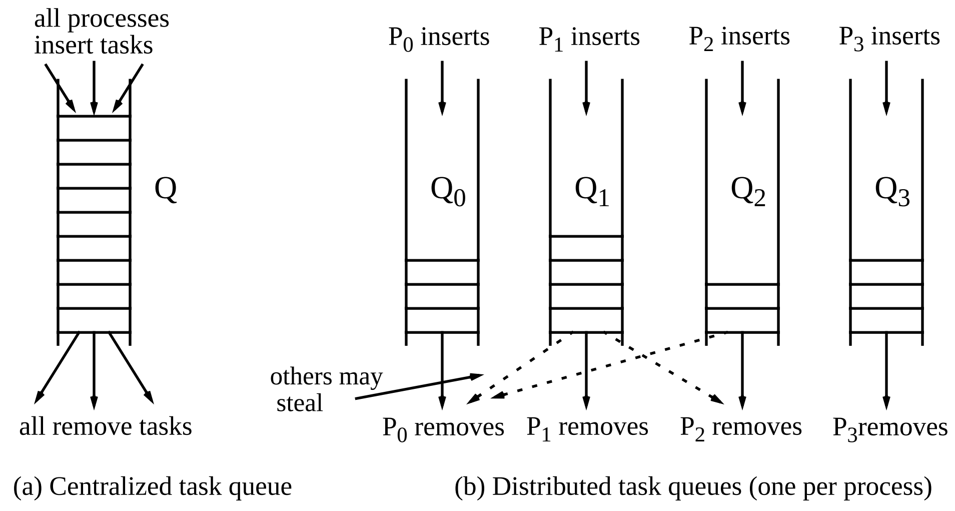


Figure 3-3 Implementing a dynamic task pool with a system of task queues.

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 [DS68,CM88]. Task queues are used in both a shared address space, where the queues are shared data structures that are manipulated using locks, as well as with explicit message passing where the owners of queues service requests for them.

While dynamic techniques generally provide good load balancing despite unpredictability or environmental conditions, they make task management 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 given both application and environment.

Determining the Granularity of Tasks

If there are no load imbalances due to dependences among tasks (for example, if all tasks to be executed are available 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 *granularity* we mean the amount of work associated with a task, 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 (there are 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 tradeoffs are clearer.

Task Granularity with Dynamic Task Queueing: 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 subtasks access the same logically shared data.

Task Granularity with Static Assignment: With static assignment, it is less clear what one should call 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 think of 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-queueing, 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 due to mutual exclusion or dependences among tasks—we must be careful about how we assign tasks and also 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 necessary; 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 and 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, tradeoffs 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, i.e. arrange the computation so 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 also less frequent. Suppose each process adds a task to the queue, then searches the queue for another task with a particular characteristic, and then remove this latter task from the queue. The task insertion and deletion may need to be mutually exclusive, 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 non-mutually-exclusive code 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 have to modify, as for the tasks we search through in the task queue, we can check without locking, and then lock and re-check (to ensure the state hasn't changed) within the critical section only if the first check returns the appropriate condition. Also, instead of using a single lock for the entire queue, we can use a lock per queue element so elements in different parts of the queue can be inserted or deleted in parallel (without serialization). As with event synchronization, the correct tradeoffs 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:

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

In general, the different aspects of balancing the workload are the responsibility of software: There is not much an architecture can do 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). Task queues have two important architectural implications. First, an access to a remote task queue is usually a probe or query, involving a small amount of data transfer and perhaps mutual exclusion. The more efficiently fine-grained communication and low-overhead, mutually exclusive access to data are supported, 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, so there is more incentive to use them instead of conservative barriers and hence achieve better load balance.

3.2.2 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 tradeoff with load balance is reducing interprocessor communication. Decomposing a problem into multiple tasks usually means that there will be communication among tasks. If these tasks are assigned to different processes, we incur communication among processes and hence processors. We focus here on reducing communication that is *inherent* to the parallel program—i.e. one process produces data that another needs—while still preserving load balance, retaining our view of the machine as a set of cooperating processors. We will see in Section 3.3 that in a real system communication occurs for other reasons as well.

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 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 that access the same data or need to communicate a lot 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 for load balancing and communication volume in practice 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—for example, the grid representing an ocean cross-section in Ocean, the space containing a galaxy in Barnes-Hut, or the image for graphics or video applications—then it is often the case that a point in the domain either requires information directly from only a small localized region around that point, or requires long-range information but the requirements fall off with distance from the point. We saw an example of the latter in Barnes-Hut. For the former, algorithms for motion estimation in video encoding and decoding examine only areas of a scene that are close to the current pixel, and 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, while 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 perimeter to area in two dimensions. Like load balance in data parallel computation, it can be reduced by either increasing the data set size (n^2 in the figure) or reducing the number of processors (p).

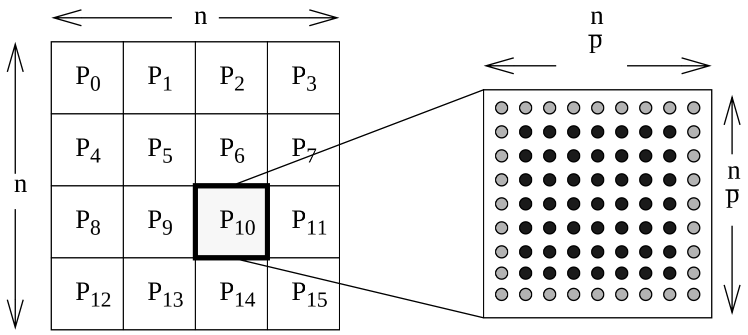


Figure 3-4 The perimeter to area relationship of communication to computation in a two-dimensional domain decomposition.

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 $\frac{n^2}{p}$ points, while communication is proportional to the number of lighter, boundary points, which is $4\frac{n}{\sqrt{p}}$.

Of course, the ideal shape for 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 chose to partition the grid into blocks of contiguous rows. Figure 3-5 shows that partitioning the grid into square-like 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 exer-

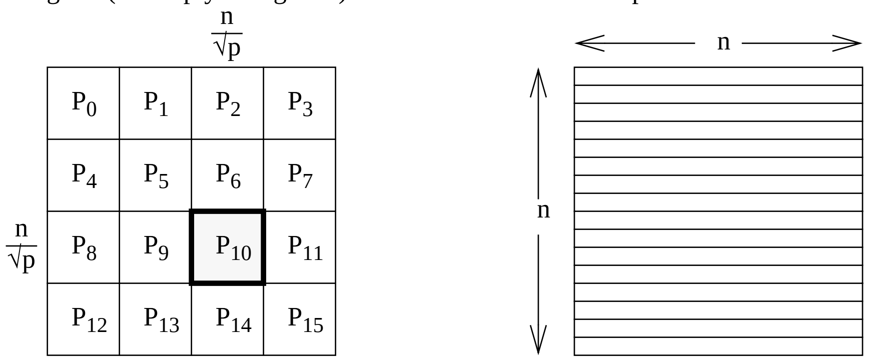


Figure 3-5 Choosing among domain decompositions for a simple nearest-neighbor computation on a regular two-dimensional grid.

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 $\frac{4 \times n / \sqrt{p}}{n^2 / p}$ or $\frac{4 \times \sqrt{p}}{n}$, while 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.

cise, 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? There are several ways, 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 on the input presented to the program at runtime, 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. In data mining, we may divide the database of transactions statically among processors, but a balanced assignment of itemsets to processes requires some analysis since the work associated with different itemsets is not equal. A simple static assignment of itemsets as well as the database by inspection keeps communication low, but does not provide load balance.

- *Semi-statically* (with periodic repartitioning). This was discussed earlier for applications like Barnes-Hut whose characteristics change with time but slowly. Domain decomposition is still important to reduce communication, and we will see the profiling-based Barnes-Hut example in Section 3.6.2.
- *Statically or semi-statically, 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 decompose the image domain just 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. Processes then steal rays (pixels) or group of rays for load balancing. The initial domain decomposition is useful because rays shot through adjacent pixels tend to access much of the same scene data.

Of course, partitioning into a contiguous subdomain per processor is not always appropriate for high performance in all applications. We shall see examples in matrix factorization in Exercise 3.3, and in a radiosity application from computer graphics in Chapter 4. Different phases of the same application may also call for different decompositions. 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 non-local queues.

In addition to reducing communication volume, it is also important to keep communication balanced among processors, not just computation. Since communication is expensive, imbalances in communication can translate directly to imbalances in execution time among processors. Overall, whether tradeoffs 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:

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

What we have done in this expression is separated out communication from work. Work now means 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 latencies and bandwidths of communication are worth spending extra money for (see Exercise 3.8); 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 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 much attention to the underlying architecture.

3.2.3 Reducing the Extra Work

The discussion of domain decomposition above shows 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 discussed above as an example of static partitioning by analysis. The 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 number of edges that cross partition boundaries is minimized and the computation is also load balanced. Many techniques have been developed for this; the ones that result in a better balance between load balance and communication require more time to compute the graph partitioning. We shall see another example in the Barnes-Hut case study in Section 3.6.2.

Another example of extra work is computing data values redundantly rather than having one process compute them and communicate them to the others, which may be a favorable tradeoff 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—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, packing/unpacking data to and from communication messages—involve extra work as well. For example, the 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 have processes be created and terminated as parallel sections of code are encountered by a single main thread of computation (a *fork-join* approach, which is sometimes used with lightweight threads instead of processes). In Data Mining, we will see that substantial extra work done to transform the database pays off in reducing communication, synchronization, and expensive input/output activity.

We must consider the tradeoffs between extra work, load balance, and communication carefully when making our partitioning decisions. The architecture can help reduce extra work by making communication and task management more efficient, and hence reducing the need for extra work. Based only on these algorithmic issues, the speedup limit can now be refined to:

$$\text{Speedup}_{problem}(p) \le \frac{\text{Sequential Work}}{max\ (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})} \qquad \textbf{(EQ 3.1)}$$

3.2.4 Summary

The analysis of parallel algorithms requires a characterization of a multiprocessor and a characterization of the parallel algorithm. Historically, the analysis of parallel algorithms has focussed on algorithmic aspects like partitioning, and has not taken 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 [FoW78]. 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 above speedup expression (Equation 3.1), and work is treated simply as instructions executed:

$$\text{Speedup-PRAM}_{problem}(p) \le \frac{\text{Sequential Instructions}}{max\ (\text{Instructions} + \text{Synch Wait Time} + \text{Extra Instructions})}\ . \qquad \textbf{(EQ 3.2)}$$

A natural way to think of a PRAM machine is as a shared address space machine in which all data access is free. The performance factors that matter in parallel algorithm analysis using the PRAM are load balance 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. 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 [Val90, CKP+93]. We will return to this issue after we have obtained a better understanding of communication costs.

The treatment of communication costs in the above discussion has been limited for two reasons when it comes to dealing with real systems. First, communication inherent to the parallel program and its partitioning is not the only form of communication that is important: There may be substantial non-inherent or *artifactual* communication 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 the equations above is not only determined by the amount of communication caused, whether inherent or artifactual, but also by the costs of the basic communication operations in the machine, the structure of the communication in the program, and how the two interact. 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, and hence open up the communication cost term, we first need a deeper understanding of some critical interactions of architectures with software.

3.3 Data Access and Communication in a Multi-Memory System

In our discussion of partitioning, we have viewed a multiprocessor as a collection of cooperating processors. However, multiprocessor systems are also multi-memory, multi-cache systems, and the role of these components is essential to performance. The role is essential regardless of programming model, though the latter may influence what the specific performance issues are. The rest of the performance issues for parallel programs have primarily to do with accessing data in this multi-memory system, so it is useful for us to now take a different view of a multiprocessor.

3.3.1 A Multiprocessor as an Extended Memory Hierarchy

From an individual processor’s perspective, we can view all the memory of the machine, including the caches of other processors, as forming levels of an extended memory hierarchy. The communication architecture glues together the parts of the hierarchy that are on different nodes. Just as interactions with levels of the memory hierarchy (for example cache size, associativity, block size) can cause the transfer of more data between levels than is inherently necessary for the program in a uniprocessor system, so also interactions with the extended memory hierarchy can cause more communication (transfer of data across the network) in multiprocessors than is inherently necessary to satisfy the processes in a parallel program. Since communication is expensive, it is particularly important that we exploit data locality in the extended hierarchy, both to improve node performance and to reduce the extra communication between nodes.

Even in uniprocessor systems, a processor's performance depends heavily on the performance of the memory hierarchy. Cache effects are so important that it hardly makes sense to talk about performance without taking caches into account. We can look at the performance of a processor in terms of the time needed to complete a program, which has two components: the time the processor is busy executing instructions and the time it spends waiting for data from the memory system.

$$\mathrm{Time}_{prog}(1) = Busy(1) + DataAccess(1) \qquad \textbf{(EQ 3.3)}$$

As architects, we often normalize this formula by dividing each term by the number of instructions and measuring time in clock cycles. We then have a convenient, machine-oriented metric of performance, cycles per instruction (CPI), which is composed of an ideal CPI plus the average number of stall cycles per instruction. On a modern microprocessor capable of issuing four instructions per cycle, dependences within the program might limit the average issue rate to, say, 2.5 instructions per cycle, or an ideal CPI of 0.4. If only 1% of these instructions cause a cache miss, and a cache miss causes the processor to stall for 80 cycles, on average, then these stalls will account for an additional 0.8 cycles per instruction. The processor will be busy doing "useful" work only one-third of its time! Of course, the other two-thirds of the time is in fact useful. It is the time spent communicating with memory to access data. Recognizing this data access cost, we may elect to optimize either the program or the machine to perform the data access more efficiently. For example, we may change how we access the data to enhance temporal or spatial locality or we might provide a bigger cache or latency tolerance mechanisms.

An idealized view of this extended hierarchy would be a hierarchy of local caches connected to a single "remote" memory at the next level. In reality the picture is a bit more complex. Even on machines with centralized shared memories, beyond the local caches is a multi-banked memory as well as the caches of other processors. With physically distributed memories, a part of the main memory too is local, a larger part is remote, and what is remote to one processor is local to another. The difference in programming models reflects a difference in how certain levels of the hierarchy are managed. We take for granted that the registers in the processor are managed explicitly, but by the compiler. We also take for granted that the first couple of levels of caches are managed transparently by the hardware. In the shared address space model, data movement between the remote level and the local node is managed transparently as well. The message passing model has this movement managed explicitly by the program. Regardless of the management, levels of the hierarchy that are closer to the processor provide higher bandwidth and lower latency access to data. We can improve data access performance either by improving the architecture of the extended memory hierarchy, or by improving the locality in the program.

Exploiting locality exposes a tradeoff with parallelism similar to reducing communication. Parallelism may cause more processors to access and draw toward them the same data, while locality from a processor's perspective desires that data stay close to it. A high performance parallel program needs to obtain performance out of each individual processor—by exploiting locality in the extended memory hierarchy—as well as being well-parallelized.

3.3.2 Artifactual Communication in the Extended Memory Hierarchy

Data accesses that are not satisfied in the local (on-node) portion of the extended memory hierarchy generate communication. Inherent communication can be seen as part of this: The data moves from one processor, through the memory hierarchy in some manner, to another processor, regardless of whether it does this through explicit messages or reads and writes. However, the amount of communication that occurs in an execution of the program is usually greater than the inherent interprocess communication. The additional communication is an artifact of how the program is actually implemented and how it interacts with the machine's extended memory hierarchy. There are many sources of this *artifactual* communication:

Poor allocation of data. Data accessed by one node may happen to be allocated in the local memory of another. Accesses to remote data involve communication, even if the data is not updated by other nodes. Such transfer can be eliminated by a better assignment or better distribution of data, or reduced by replicating the data when it is accessed.

Unnecessary data in a transfer. More data than needed may be communicated in a transfer. For example, a receiver may not use all the data in a message: It may have been easier to send extra data conservatively than determine exactly what to send. Similarly, if data is transferred implicitly in units larger than a word, e.g. cache blocks, part of the block may not be used by the requester. This artifactual communication can be eliminated with smaller transfers.

Unnecessary transfers due to system granularities. In cache-coherent machines, data may be kept coherent at a granularity larger than a single word, which may lead to extra communication to keep data coherent as we shall see in later chapters.

Redundant communication of data. Data may be communicated multiple times, for example, every time the value changes, but only the last value may actually be used. On the other hand, data may be communicated to a process that already has the latest values, again because it was too difficult to determine this.

Finite replication capacity. The capacity for replication on a node is finite—whether it be in the cache or the main memory—so data that has been already communicated to a process may be replaced from its local memory system and hence need to be transferred again even if it has not since been modified.

In contrast, inherent communication is that which would occur with unlimited capacity for replication, transfers as small as necessary, and perfect knowledge of what logically shared data has been updated. We will understand some of the sources of artifactual communication better when we get deeper into architecture. Let us look a little further at the last source of artifactual communication—finite replication capacity—which has particularly far-reaching consequences.

Artifactual Communication and Replication: The Working Set Perspective

The relationship between finite replication capacity and artifactual communication is quite fundamental in parallel systems, just like the relationship between cache size and memory traffic in uniprocessors. It is almost inappropriate to speak of the amount of communication without reference to replication capacity. The extended memory hierarchy perspective is useful in viewing this relationship. We may view our generic multiprocessor as a hierarchy with three levels: Local cache is inexpensive to access, local memory is more expensive, and any remote memory is much more expensive. We can think of any level as a cache, whether it is actually managed like a hardware cache or by software. Then, we can then classify the "misses" at any level, which generate traffic to the next level, just as we do for uniprocessors. A fraction of the traffic at any level is *cold-start*, resulting from the first time data is accessed by the processor. This component, also called *compulsory* traffic in uniprocessors, is independent of cache size. Such cold start misses are a concern in performing cache simulations, but diminish in importance as programs run longer. Then there is traffic due to *capacity* misses, which clearly decrease with increases in cache size. A third fraction of traffic may be *conflict* misses, which are reduced by greater associativity in the replication store, a greater number of blocks, or changing the data access pattern. These three types of misses or traffic are called the three C's in uniprocessor architecture (cold start or compulsory, capacity, conflict). The new form of traffic in multiprocessors is a fourth C, a *communication* miss, caused by the inherent communication between processors or some of the sources of artifactual communication discussed above. Like cold start misses, communication misses do not diminish with cache size. Each of these components of traffic may be helped or hurt by large granularities of data transfer depending on spatial locality.

If we were to determine the traffic resulting from each type of miss for a parallel program as we increased the replication capacity (or cache size), we could expect to obtain a curve such as shown in Figure 3-6. The curve has a small number of knees or points of inflection. These knees correspond to the *working sets* of the algorithm relevant to that level of the hierarchy.[1] For the first-level cache, they are the working sets of the algorithm itself; for others they depend on how references have been filtered by other levels of the hierarchy and on how the levels are managed. We speak of this curve for a first-level cache (assumed fully associative with a one-word block size) as the working set curve for the algorithm.

Traffic resulting from any of these types of misses may cause communication across the machine's interconnection network, for example if the backing storage happens to be in a remote node. Similarly, any type of miss may contribute to local traffic and data access cost. Thus, we might expect that many of the techniques used to reduce artifactual communication are similar to those used to exploit locality in uniprocessors, with some additional ones. Inherent communication misses almost always generate actual communication in the machine (though sometimes they may not if the data needed happens to have become local in the meanwhile, as we shall see), and can only be reduced by changing the sharing patterns in the algorithm. In addition, we are strongly motivated to reduce the artifactual communication that arises either due to transfer size or due to limited replication capacity, which we can do by exploiting spatial and temporal locality in a process's data accesses in the extended hierarchy. Changing the orchestration and assignment can dramatically change locality characteristics, including the shape of the working set curve.

For a given amount of communication, its cost as seen by the processor is also affected by how the communication is structured. By structure we mean whether messages are large or small, how

1. The working set model of program behavior [Den68] is based on the temporal locality exhibited by the data referencing patterns of programs. Under this model, a program—or a process in a parallel program—has a set of data that it reuses substantially for a period of time, before moving on to other data. The shifts between one set of data and another may be abrupt or gradual. In either case, there is at most times a "working set" of data that a processor should be able to maintain in a fast level of the memory hierarchy, to use that level effectively.

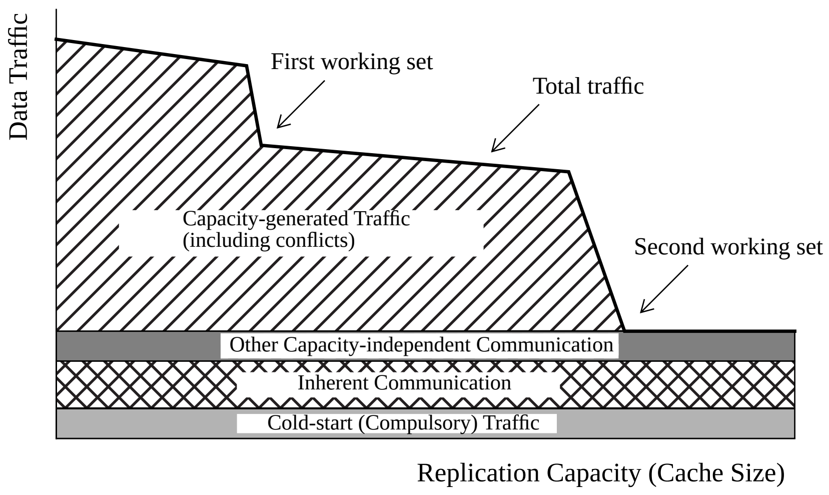


Figure 3-6 The data traffic between a cache (replication store) and the rest of the system, and its components as a function of cache size.

The points of inflection in the total traffic curve indicate the working sets of the program.

bursty the communication is, whether communication cost can be overlapped with other computation or communication—all of which are addressed in the orchestration step—and how well the communication patterns match the topology of the interconnection network, which is addressed in the mapping step. Reducing the amount of communication—inherent or artifactual—is important because it reduces the demand placed on both the system and the programmer to reduce cost. Having understood the machine as an extended hierarchy and the major issues this raises, let us see how to address these architecture-related performance issues at the next level in software; that is, how to program for performance once partitioning issues are resolved.

3.4 Orchestration for Performance

We begin by discussing how we might exploit temporal and spatial locality to reduce the amount of artifactual communication, and then move on to structuring communication—inherent or artifactual—to reduce cost.

3.4.1 Reducing Artifactual Communication

In the message passing model both communication and replication are explicit, so even artifactual communication is explicitly coded in program messages. In a shared address space, artifactual communication is more interesting architecturally since it occurs transparently due to interactions between the program and the machine organization; in particular, the finite cache size and the granularities at which data are allocated, communicated and kept coherent. We therefore use a shared address space to illustrate issues in exploiting locality, both to improve node performance and to reduce artifactual communication.

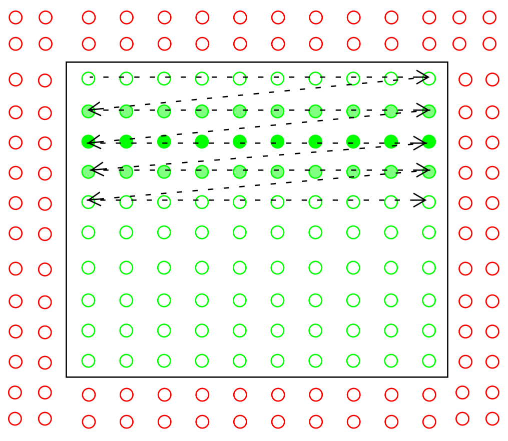
Exploiting Temporal Locality

A program is said to exhibit temporal locality if tends to access the same memory location repeatedly in a short time-frame. Given a memory hierarchy, the goal in exploiting temporal locality is to structure an algorithm so that its working sets map well to the sizes of the different levels of the hierarchy. For a programmer, this typically means keeping working sets small without losing performance for other reasons. Working sets can be reduced by several techniques. One is the same technique that reduces inherent communication—assigning tasks that tend to access the same data to the same process—which further illustrates the relationship between communication and locality. Once assignment is done, a process's assigned computation can be organized so that tasks that access the same data are scheduled close to one another in time, and so that we reuse a set of data as much as possible before moving on to other data, rather than moving on and coming back.

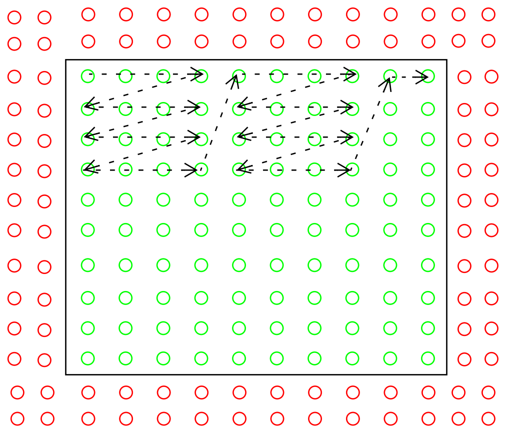
When multiple data structures are accessed in the same phase of a computation, we must decide which are the most important candidates for exploiting temporal locality. Since communication is more expensive than local access, we might prefer to exploit temporal locality on nonlocal rather than local data. Consider a database application in which a process wants to compare all its records of a certain type with all the records of other processes. There are two choices here: (i) for each of its own records, the process can sweep through all other (nonlocal) records and compare, and (ii) for each nonlocal record, the process can sweep through its own records and compare. The latter exploits temporal locality on nonlocal data, and is therefore likely to yield better overall performance.

Example 3-1 To what extent is temporal locality exploited in the equation solver kernel. How might the temporal locality be increased?

Answer The equation solver kernel traverses only a single data structure. A typical grid element in the interior of a process's partition is accessed at least five times by that process during each sweep: at least once to compute its own new value, and once each to compute the new values of its four nearest neighbors. If a process sweeps through its partition of the grid in row-major order (i.e. row by row and left to right within each row, see Figure 3-7(a)) then reuse of *A[i,j]* is guaranteed across the updates of the three elements in the same row that touch it: *A[i,j-1]*, *A[i,j]* and *A[i,j+1]*. However, between the times that the new values for *A[i,j]* and *A[i+1,j]* are computed, three whole subrows of elements in that process's partition are accessed by that process. If the three subrows don't fit together in the cache, then *A[i,j]* will no longer be in the cache when it is accessed again to compute *A[i+1,j]*. If the backing store for these data in the cache is nonlocal, artifactual communication will result. The problem can be fixed by changing the order in which elements are computed, as shown in Figure 3-7(b). Essentially, a process proceeds left-to-right not for the length of a whole subrow of its partition, but only a certain length *B*, before it moves on to the corresponding portion of the next subrow. It performs its sweep in sub-sweeps over *B*-by-*B* blocks of its partition. The



(a) Unblocked access pattern in a sweep



(b) Blocked access pattern with B=4

Figure 3-7 Blocking to exploit temporal locality in the equation solver kernel.

The figures show the access patterns for a process traversing its partition during a sweep, with the arrow-headed lines showing the order in which grid points are updated. Updating the subrow of bold elements requires accessing that subrow as well as the two subrows of shaded elements. Updating the first element of the next (shaded) subrow requires accessing the first element of the bold subrow again, but these three whole subrows have been accessed since the last time that first bold element was accessed.

block size *B* is chosen so that at least three *B*-length rows of a partition fit in the cache.

This technique of structuring computation so that it accesses a subset of data that fits in a level of the hierarchy, uses those data as much as possible, and then moves on to the next such set of data, is called *blocking*. In the particular example above, the reduction in miss rate due to blocking is only a small constant factor (about a factor of two). The reduction is only seen when a subrow of a process's partition of a grid itself does not fit in the cache, so blocking is not always useful. However, blocking is used very often in linear algebra computations like matrix multiplication or matrix factorization, where $O(n^{k+1})$ computation is performed on a data set of size $O(n^k)$, so each data item is accessed $O(n)$ times. Using blocking effectively with *B*-by-*B* blocks in these cases can reduce the miss rate by a factor of *B*, as we shall see in Exercise 3.5, which is particularly important since much of the data accessed is nonlocal. Not surprisingly, many of the same types of restructuring are used to improve temporal locality in a sequential program as well; for example, blocking is critical for high performance in sequential matrix computations. Techniques for temporal locality can be used at any level of the hierarchy where data are replicated—including main memory—and for both explicit or implicit replication.

Since temporal locality affects replication, the locality and referencing patterns of applications have important implications for determining which programming model and communication abstraction a system should support. We shall return to this issue in Section 3.7. The sizes and scaling of working sets have obvious implications for the amounts of replication capacity needed at different levels of the memory hierarchy, and the number of levels that make sense in this hierarchy. In a shared address space, together with their compositions (whether they hold local or remote data or both), working set sizes also help determine whether it is useful to replicate communicated data in main memory as well or simply rely on caches, and if so how this should be done. In message passing, they help us determine what data to replicate and how to manage the replication so as not to fill memory with replicated data. Of course it is not only the working sets of individual applications that matter, but those of the entire workloads and the operating system that run on the machine. For hardware caches, the size of cache needed to hold a working set depends on its organization (associativity and block size) as well.

Exploiting Spatial Locality

A level of the extended memory hierarchy exchanges data with the next level at a certain *granularity* or transfer size. This granularity may be fixed (for example, a cache block or a page of main memory) or flexible (for example, explicit user-controlled messages or user-defined objects). It usually becomes larger as we go further away from the processor, since the latency and fixed startup overhead of each transfer becomes greater and should be amortized over a larger amount of data. To exploit a large *granularity of communication or data transfer*, we should organize our code and data structures to exploit spatial locality.[1] Not doing so can lead to artifactual communication if the transfer is to or from a remote node and is implicit (at some fixed granularity), or to more costly communication even if the transfer is explicit (since either smaller messages may have to be sent or the data may have to be made contiguous before they are sent). As in uniprocessors, poor spatial locality can also lead to a high frequency of TLB misses.

In a shared address space, in addition to the granularity of communication, artifactual communication can also be generated by mismatches of spatial locality with two other important granularities. One is the *granularity of allocation*, which is the granularity at which data are allocated in the local memory or replication store (e.g. a page in main memory, or a cache block in the cache). Given a set of data structures, this determines the granularity at which the data can be distributed among physical main memories; that is, we cannot allocate a part of a page in one node's memory and another part of the page in another node's memory. For example, if two words that are mostly accessed by two different processors fall on the same page, then unless data are replicated in multiple main memories that page may be allocated in only one processor's local memory; capacity or conflict cache misses to that word by the other processor will generate communication. The other important granularity is the *granularity of coherence*, which we will discuss in Chapter 5. We will see that unrelated words that happen to fall on the same unit of coherence in a coherent shared address space can also cause artifactual communication.

The techniques used for all these aspects of spatial locality in a shared address space are similar to those used on a uniprocessor, with one new aspect: We should try to keep the data accessed by a given processor close together (contiguous) in the address space, and unrelated data accessed by different processors apart. Spatial locality issues in a shared address space are best examined in the context of particular architectural styles, and we shall do so in Chapters 5 and 8. Here, for illustration, we look at one example: how data may be restructured to interact better with the granularity of allocation in the equation solver kernel.

1. The principle of spatial locality states that if a given memory location is referenced now, then it is likely that memory locations close to it will be referenced in the near future.It should be clear that what is called spatial locality at the granularity of individual words can also be viewed as temporal locality at the granularity of cache blocks; that is, if a cache block is accessed now, then it (another datum on it) is likely to be accessed in the near future.

Example 3-2 Consider a shared address space system in which main memory is physically distributed among the nodes, and in which the granularity of allocation in main memory is a page (4 KB, say). Now consider the grid used in the equation solver kernel. What is the problem created by the granularity of allocation, and how might it be addressed?

Answer The natural data structure with which to represent a two-dimensional grid in a shared address space, as in a sequential program, is a two-dimensional array. In a typical programming language, a two-dimensional array data structure is allocated in either a “row-major” or “column-major” way.[1] The gray arrows in Figure 3-8(a) show the contiguity of virtual addresses in a row-major allocation, which is the one we assume. While a two-dimensional shared array has the programming advantage

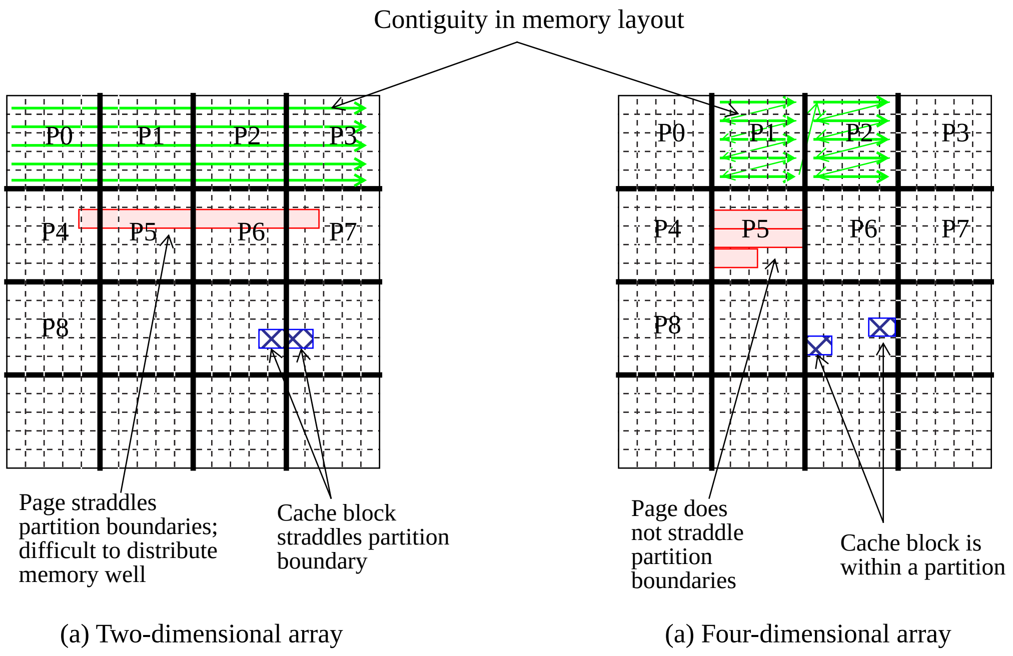

Contiguity in memory layout
P0
P1
P2
P3
P4
P5
P6
P7
P8
Page straddles partition boundaries; difficult to distribute memory well
Cache block straddles partition boundary
Page does not straddle partition boundaries
Cache block is within a partition
(a) Two-dimensional array
(a) Four-dimensional array

Figure 3-8 Two-dimensional and Four-dimensional arrays used to represent a two-dimensional grid in a shared address space.

of being the same data structure used in a sequential program, it interacts poorly

1. Consider the array as being a two-dimensional grid, with the first dimension specifying the row number in the grid and the second dimension the column number. Row-major allocation means that all elements in the first row are contiguous in the virtual address space, followed by all the elements in the second row, etc. The C programming language, which we assume here, is a row-major language (FORTRAN, for example, is column-major).

with the granularity of allocation on a machine with physically distributed memory (and with other granularities as well, as we shall see in Chapter 5).

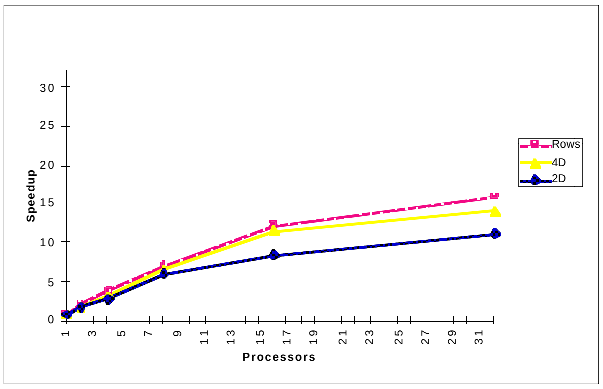
Consider the partition of processor *P5* in Figure 3-8(a). An important working set for the processor is its entire partition, which it streams through in every sweep and reuses across sweeps. If its partition does not fit in its local cache hierarchy, we would like it to be allocated in local memory so that the misses can be satisfied locally. The problem is that consecutive subrows of this partition are not contiguous with one another in the address space, but are separated by the length of an entire row of the grid (which contains subrows of other partitions). This makes it impossible to distribute data appropriately across main memories if a subrow of a partition is either smaller than a page, or not a multiple of the page size, or not well aligned to page boundaries. Subrows from two (or more) adjacent partitions will fall on the same page, which at best will be allocated in the local memory of one of those processors. If a processor's partition does not fit in its cache, or it incurs conflict misses, it may have to communicate every time it accesses a grid element in its own partition that happens to be allocated nonlocally.

The solution in this case is to use a higher-dimensional array to represent the two-dimensional grid. The most common example is a four-dimensional array, in which case the processes are arranged conceptually in a two-dimensional grid of partitions as seen in Figure 3-8. The first two indices specify the partition or process being referred to, and the last two represent the subrow and subcolumn numbers within that partition. For example, if the size of the entire grid is 1024-by-1024 elements, and there are 16 processors, then each partition will be a subgrid of size $\frac{1024}{\sqrt{16}} - \mathrm{by} - \frac{1024}{\sqrt{16}}$ or 256-by-256 elements. In the four-dimensional representation, the array will be of size 4-by-4-by-256-by-256 elements. The key property of these higher-dimensional representations is that each processor's 256-by-256 element partition is now contiguous in the address space (see the contiguity in the virtual memory layout in Figure 3-8(b)). The data distribution problem can now occur only at the end points of entire partitions, rather than of each subrow, and does not occur at all if the data structure is aligned to a page boundary. However, it is substantially more complicated to write code using the higher-dimensional arrays, particularly for array indexing of neighboring process's partitions in the case of the nearest-neighbor computation.

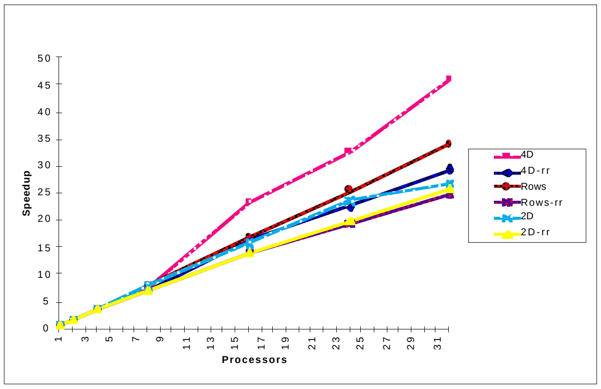
More complex applications and data structures illustrate more significant tradeoffs in data structure design for spatial locality, as we shall see in later chapters.

The spatial locality in processes' access patterns, and how they scale with the problem size and number of processors, affects the desirable sizes for various granularities in a shared address space: allocation, transfer and coherence. It also affects the importance of providing support tailored toward small versus large messages in message passing systems. Amortizing hardware and transfer cost pushes us toward large granularities, but too large granularities can cause performance problems, many of them specific to multiprocessors. Finally, since conflict misses can generate artifactual communication when the backing store is nonlocal, multiprocessors push us toward higher associativity for caches. There are many cost, performance and programmability tradeoffs concerning support for data locality in multiprocessors—as we shall see in subsequent chapters—and our choices are best guided by the behavior of applications.

Finally, there are often interesting tradeoffs among algorithmic goals such as minimizing inherent communication, implementation issues, and architectural interactions that generate artifactual communication, suggesting that careful examination of tradeoffs is needed to obtain the best performance on a given architecture. Let us illustrate using the equation solver kernel.



(a) Ocean with 514-by-514 grids



(b) Equation solver kernel with 12K-by-12K grid

Figure 3-9 The impact of data structuring and spatial data locality on performance.

All measurements are on the SGI Origin2000. 2D and 4D imply two- and four-dimensional data structures, respectively, with subblock assignment. Rows uses the two-dimensional array with strip assignment into chunks of rows. In the right graph, the postfix rr means that pages of data are distributed round-robin among physical memories. Without rr, it means that pages are placed in the local memory of the processor to which their data is assigned, as far as possible. We see from (a) that the rowwise strip assignment outperforms the 2D strip assignment due to spatial locality interactions with long cache blocks (128 bytes on the Origin2000), and even a little better than the 4D array block assignment due to poor spatial locality in the latter in accessing border elements at column-oriented partition boundaries. The right graph shows that in all partitioning schemes proper data distribution in main memory is important to performance, though least successful for the 2D array subblock partitions. In the best case, we see superlinear speedups once there are enough processors that the size of a processor's partition of the grid (its important working set) fits into its cache.

Example 3-3 Given the performance issues discussed so far, should we choose to partition the equation solver kernel into square-like blocks or into contiguous strips of rows?

Answer If we only consider inherent communication, we already know that a block domain decomposition is better than partitioning into contiguous strips of rows (see Figure 3-5 on page 142). However, a strip decomposition has the advantage that it keeps a partition wholly contiguous in the address space even with the simpler, two-dimensional array representation. Hence, it does not suffer problems related to the interactions of spatial locality with machine granularities, such as the granularity of allocation as discussed above. This particular interaction in the block case can of course be solved by using a higher-dimensional array representation. However, a more difficult interaction to solve is with the granularity of communication. In a subblock assignment, consider a neighbor element from another partition at a column-oriented partition boundary. If the granularity of communication is large, then when a process references this element from its neighbor's partition it will fetch not only that element but also a number of other elements that are on the same unit of communication. These other elements are not neighbors of the fetching process's partition, regardless of whether a two-dimensional or four-dimensional representation is used, so they are useless and waste communication bandwidth (see Figure 3-10). With a partitioning into strips of rows, a referenced element still

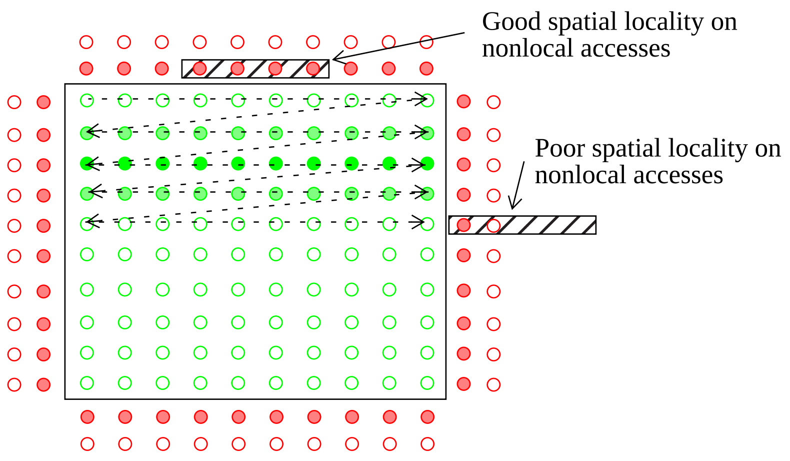

Good spatial locality on nonlocal accesses
Poor spatial locality on nonlocal accesses

Figure 3-10 Spatial locality in accesses to nonlocal data in the equation solver kernel.

The shaded points are the nonlocal points that the processor owning the partition accesses. The hatched rectangles are cache blocks, showing good spatial locality along the row boundary but poor along the column.

causes other elements from its row to be fetched, but now these elements are indeed neighbors of the fetching process's partition. They are useful, and in fact the large granularity of communication results in a valuable prefetching effect. Overall, there are many combinations of application and machine parameters for which the performance losses in block partitioning owing to artifactual communication will dominate the performance benefits from reduced inherent communication, so that a strip partitioning will perform better than a block partitioning on a real system. We might imagine that it should most often perform better when a two-dimensional array is used in the block case, but it may also do so in some cases when a four-dimensional array is used (there is not motivation to use a four-dimensional array with a strip partitioning). Thus, artifactual communication may cause us to go back and revise our partitioning method from block to strip. Figure 3-9(a) illustrates this effect for the Ocean application, and Figure 3-9(b) uses the equation solver kernel with a larger grid size to also illustrate the impact of data placement. Note that a strip decomposition into columns rather than rows will yield the worst of both worlds when data are laid out in memory in row-major order.

3.4.2 Structuring Communication to Reduce Cost

Whether communication is inherent or artifactual, how much a given communication-to-computation ratio contributes to execution time is determined by how the communication is organized or structured into messages. A small communication-to-computation ratio may have a much greater impact on execution time than a large ratio if the structure of the latter interacts much better with the system. This is an important issue in obtaining good performance from a real machine, and is the last major performance issue we examine. Let us begin by seeing what the structure of communication means.

In Chapter 1, we introduced a model for the cost of communication as seen by a processor given a frequency of program initiated communication operations or messages (initiated explicit messages or loads and stores). Combining equations 1.5 and 1.6, that model for the cost C is:

$$C = freq \times \left(Overhead + Delay + \frac{Length}{Bandwidth} + Contention - Overlap \right)$$

or

$$C = f \times \left(o + l + \frac{n_c / m}{B} + t_c - overlap \right) \qquad \textbf{(EQ 3.4)}$$

where

f is the frequency of communication messages in the program.

n_c is the total amount of data communicated by the program.

m is the number of message, so n_c/m is the average length of a message.

o is the combined overhead of handling the initiation and reception of a message at the sending and receiving processors, assuming no contention with other activities.

l is the delay for the first bit of the message to reach the destination processor or memory, which includes the delay through the assists and network interfaces as well as the network latency or transit latency in the network fabric itself (a more detailed model might separate the two out). It assumes no contention.

B is the point-to-point bandwidth of communication afforded for the transfer by the communication path, excluding the processor overhead; i.e. the rate at which the rest of the message data arrives at the destination after the first bit, assuming no contention. It is the inverse of the overall occupancy discussed in Chapter 1. It may be limited by the network links, the network interface or the communication assist.

t_c is the time induced by contention for resources with other activities.

overlap is the amount of the communication cost that can be overlapped with computation or other communication, i.e that is not in the critical path of a processor's execution.

This expression for communication cost can be substituted into the speedup equation we developed earlier (Equation 3.1 on page 144), to yield our final expression for speedup. The portion of the cost expression inside the parentheses is our cost model for a single data one-way message. It assumes the "cut-through" or pipelined rather than store-and-forward transmission of modern multiprocessor networks. If messages are round-trip, we must make the appropriate adjustments. in addition to reducing communication volume (n_c) our goals in structuring communication may include (i) reducing communication overhead ($m*o$), (ii) reducing latency ($m*l$), (iii) reducing contention ($m*t_c$), and (iv) overlapping communication with computation or other communication to hide its latency. Let us discuss programming techniques for addressing each of the issues.

Reducing Overhead

Since the overhead *o* associated with initiating or processing a message is usually fixed by hardware or system software, the way to reduce communication overhead is to make messages fewer in number and hence larger, i.e. to reduce *t*he message frequency[1] Explicitly initiated communication allows greater flexibility in specifying the sizes of messages; see the SEND primitive described in Section 2.4.6 in the previous chapter. On the other hand, implicit communication through loads and stores does not afford the program direct control, and the system must take responsibility for coalescing the loads and stores into larger messages if necessary.

Making messages larger is easy in applications that have regular data access and communication patterns. For example, in the message passing equation solver example partitioned into rows we sent an entire row of data in a single message. But it can be difficult in applications that have irregular and unpredictable communication patterns, such as Barnes-Hut or Raytrace. As we shall see in Section 3.7, it may require changes to the parallel algorithm, and extra work to determine which data to coalesce, resulting in a tradeoff between the cost of this computation and the savings in overhead. Some computation may be needed to determine what data should be sent, and the data may have to be gathered and packed into a message at the sender and unpacked and scattered into appropriate memory locations at the receiver.

Reducing Delay

Delay through the assist and network interface can be reduced by optimizing those components. Consider the network transit delay or the delay through the network itself. In the absence of contention and with our pipelined network assumption, the transit delay l of a bit through the network itself can be expressed as $h*t_h$, where h is the number of "hops" between adjacent network nodes that the message traverses, t_h is the delay or latency for a single bit of data to traverse a single network hop, including the link and the router or switch. Like o above, t_h is determined by the system, and the program must focus on reducing the f and h components of the $f*h* t_h$ delay cost. (In store-and-forward networks, t_h would be the time for the entire message to traverse a hop, not just a single bit.)

The number of hops h can be reduced by *mapping* processes to processors so that the topology of interprocess communication in the application exploits locality in the physical topology of the network. How well this can be done in general depends on application and on the structure and richness of the network topology; for example, the nearest-neighbor equation solver kernel (and the Ocean application) would map very well onto a mesh-connected multiprocessor but not onto a unidirectional ring topology. Our other example applications are more irregular in their communication patterns. We shall see several different topologies used in real machines and discuss their tradeoffs in Chapter 10.

There has been a lot of research in mapping algorithms to network topologies, since it was thought that as the number of processors p became large poor mappings would cause the latency due to the $h*t_h$ term to dominate the cost of messages. How important topology actually is in practice depends on several factors: (i) how large the t_h term is relative to the overhead o of getting a message into and out of the network, (ii) the number of processing nodes on the machine, which determines the maximum number of hops h for a given topology, and (iii) whether the machine is used to run a single application at a time in "batch" mode or is multiprogrammed among applications. It turns out that network topology is not considered as important on modern machines as it once was, because of the characteristics of the machines along all three axes: overhead dominates hop latency (especially in machines that do not provide hardware support for a shared address space), the number of nodes is usually not very large, and the machines are often

1. Some explicit message passing systems provide different types of messages with different costs and functionalities among which a program can choose.

used as general-purpose, multiprogrammed servers. Topology-oriented design might not be very useful in multiprogrammed systems, since the operating system controls resource allocation dynamically and might transparently change the mapping of processes to processors at runtime. For these reasons, the mapping step of parallelization receives considerably less attention than decomposition, assignment and orchestration. However, this may change again as technology and machine architecture evolve. Let us now look at contention, the third major cost issue in communication.

Reducing Contention

The communication systems of multiprocessors consist of many resources, including network links and switches, communication controllers, memory systems and network interfaces. All of these resources have a nonzero *occupancy*, or time for which they are occupied servicing a given transaction. Another way of saying this is that they have finite bandwidth (or rate, which is the reciprocal of occupancy) for servicing transactions. If several messages contend for a resource, some of them will have to wait while others are serviced, thus increasing message latency and reducing the bandwidth available to any single message. Resource occupancy contributes to message cost even when there is no contention, since the time taken to pass through resource is part of the delay or overhead, but it can also cause contention.

Contention is a particularly insidious performance problem, for several reasons. First, it is easy to ignore when writing a parallel program, particularly if it is caused by artifactual communication. Second, its effect on performance can be dramatic. If *p* processors contend for a resource of occupancy *x*, the first to obtain the resource incurs a latency of *x* due to that resource, while the last incurs at least *p*x*. In addition to large stall times for processors, these differences in wait time across processors can also lead to large load imbalances and synchronization wait times. Thus, the contention caused by the occupancy of a resource can be much more dangerous than just the latency it contributes in uncontended cases. The third reason is that contention for one resource can hold up other resources, thus stalling transactions that don't even need the resource that is the source of the contention. This is similar to how contention for a single-lane exit off a multi-lane highway causes congestion on the entire stretch of highway. The resulting congestion also affects cars that don't need that exit but want to keep going on the highway, since they may be stuck behind cars that do need that exit. The resulting backup of cars covers up other unrelated resources (previous exits), making them inaccessible and ultimately clogging up the highway. Bad cases of contention can quickly saturate the entire communication architecture. The final reason is related: The cause of contention is particularly difficult to identify, since the effects might be felt at very different points in the program than the original cause, particularly communication is implicit.

Like bandwidth, contention in a network can also be viewed as being of two types: at links or switches within the network, called *network contention*, and at the end-points or processing nodes, called *end-point contention*. Network contention, like latency, can be reduced by mapping processes and scheduling the communication appropriately in the network topology. End-point contention occurs when many processors need to communicate with the same processing node at the same time (or when communication transactions interfere with local memory references). When this contention becomes severe, we call that processing node or resource a *hot spot*. Let us examine a simple example of how a hot spot may be formed, and how it might be alleviated in software.

Recall the case of processes want to accumulate their partial sums into a global sum, as in our equation solver kernel. The resulting contention for the global sum can be reduced by using tree-structured communication rather than having all processes send their updates to the owning node directly. Figure 3-11 shows the structure of such many-to-one communication using a binary fan-in tree. The nodes of this tree, which is often called a software combining tree, are the participating processes. A leaf process sends its update up to its parent, which combines its children's updates with its own and sends the combined update up to its parent, and so on till the updates reach the root (the process that holds the global sum) in $\log_2 p$ steps. A similar fan-out tree can be used to send data from one to many processes. Similar tree-based approaches are used to design

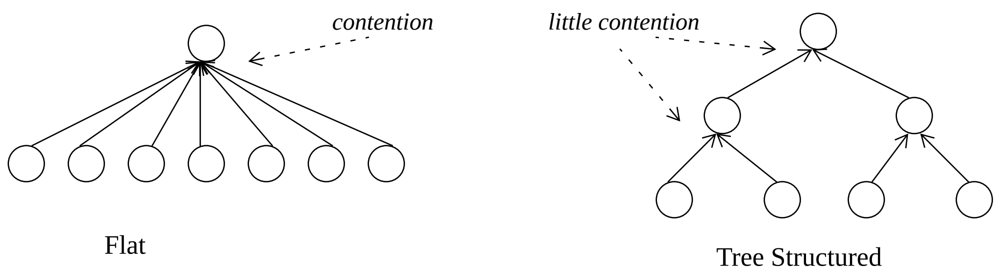


Figure 3-11 Two ways of structuring many-to-one communication: flat, and tree-structured in a binary fan-in tree.

Note that the destination processor may received up to *p-1* messages at a time in the flat case, while no processor is the destination of more than two messages in the binary tree.

scalable synchronization primitives like locks and barriers that often experience a lot of contention, as we will see in later chapters, as well as library routines for other communication patterns.

In general, two principles for alleviating contention are to avoid having too many processes communicate with the same process at the same time, and to stagger messages to the same destination so as not to overwhelm the destination or the resources along the way. Contention is often caused when communication is bursty (i.e. the program spends some time not communicating much and then suddenly goes through a burst of communication), and staggering reduces burstiness. However, this must be traded off with making messages large, which tends to increase burstiness. Finally, having discussed overhead, latency, and contention, let us look at the last of the communication cost issues.

Overlapping Communication with Computation or Other Communication

Despite efforts to reduce overhead and delay, the technology trends discussed in Chapter 1 suggest that the end-to-end the communication cost as seen by a processor is likely to remain very large in processor cycles. Already, it is in the hundreds of processor cycles even on machines that provide full hardware support for a shared address space and use high-speed networks, and is at least an order of magnitude higher on current message passing machines due to the higher overhead term *o*. If the processor were to remain idle (stalled) while incurring this cost for every word of data communicated, only programs with an extremely low ratio of communication to computation would yield effective parallel performance. Programs that communicate a lot must therefore find ways to hide the cost of communication from the process's critical path by overlapping it with computation or other communication as much as possible, and systems must provide the necessary support.

Techniques to hide communication cost come in different, often complementary flavors, and we shall spend a whole chapter on them. One approach is simply to make messages larger, thus incurring the latency of the first word but hiding that of subsequent words through pipelined transfer of the large message. Another, which we can call precommunication, is to initiate the communication much before the data are actually needed, so that by the time the data are needed they are likely to have already arrived. A third is to perform the communication where it naturally belongs in the program, but hide its cost by finding something else for the processor to do from later in the same process (computation or other communication) while the communication is in progress. A fourth, called multithreading is to switch to a different thread or process when one encounters a communication event. While the specific techniques and mechanisms depend on the communication abstraction and the approach taken, they all fundamentally require the program to have extra concurrency (also called *slackness*) beyond the number of processors used, so that independent work can be found to overlap with the communication.

Much of the focus in parallel architecture has in fact been on reducing communication cost as seen by the processor: reducing communication overhead and latency, increasing bandwidth, and providing mechanisms to alleviate contention and overlap communication with computation or other communication. Many of the later chapters will therefore devote a lot of attention to covering these issues—including the design of node to network interfaces and communication protocols and controllers that minimize both software and hardware overhead (Chapter 7, 8 and 9), the design of network topologies, primitive operations and routing strategies that are well-suited to the communication patterns of applications (Chapter 10), and the design of mechanisms to hide communication cost from the processor (Chapter 11). The architectural methods are usually expensive, so it is important that they can be used effectively by real programs and that their performance benefits justify their costs.

3.5 Performance Factors from the Processors' Perspective

To understand the impact of different performance factors in a parallel program, it is useful to look from an individual processor's viewpoint at the different components of time spent executing the program; i.e. how much time the processor spends in different activities as it executes instructions and accesses data in the extended memory hierarchy. These different components of time can be related quite directly to the software performance issues studied in this chapter, helping us relate software techniques to hardware performance. This view also helps us understand what a parallel execution looks like as a workload presented to the architecture, and will be useful when we discuss workload-driven architectural evaluation in the next chapter.

In Equation 3.3, we described the time spent executing a sequential program on a uniprocessor as the sum of the time actually executing instructions (*busy*) and the time stalled on the memory system (*data-local*), where the latter is a "non-ideal" factor that reduces performance. Figure 3-12(a) shows a profile of a hypothetical sequential program. In this case, about 80% of the execution time is spent performing instructions, which can be reduced only by improving the algorithm or the processor. The other 20% is spent stalled on the memory system, which can be improved by improving locality or the memory system.

In multiprocessors we can take a similar view, though there are more such non-ideal factors. This view cuts across programming models: for example, being stalled waiting for a receive to complete is really very much like being stalled waiting for a remote read to complete or a synchronization event to occur. If the same program is parallelized and run on a four-processor machine, the execution time profile of the four processors might look like that in Figure 3-12(b). The figure

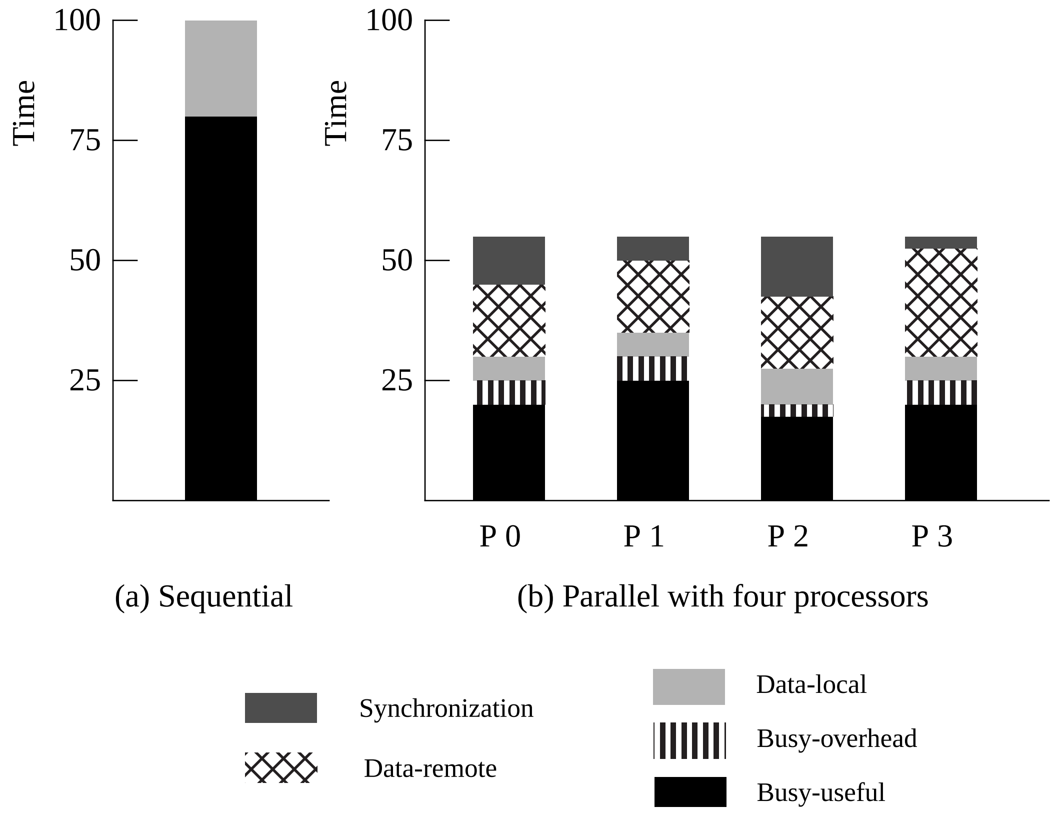


Figure 3-12 Components of execution time from the perspective of an individual processor.

assumes a global synchronization point at the end of the program, so that all processes terminate at the same time. Note that the parallel execution time (55 sec) is greater than one-fourth of the sequential execution time (100 sec); that is, we have obtained a speedup of only $\frac{100}{55}$ or 1.8 instead of the four-fold speedup for which we may have hoped. Why this is the case, and what specific software or programming factors contribute to it can be determined by examining the components of parallel execution time from the perspective of an individual processor. These are:

Busy-useful: this is the time that the processor spends executing instructions that would have been executed in the sequential program as well. Assuming a deterministic parallel program[1] that is derived directly from the sequential algorithm, the sum of the busy-useful times for all processors is equal to the busy-useful time for the sequential execution. (This is true at least for processors that execute a single instruction per cycle; attributing cycles of execution time to busy-useful or other categories is more complex when processors issue multiple instructions per cycle, since different cycles may execute different numbers of instructions and a given stall cycle may be attributable to different causes. We will discuss this further in Chapter 11 when we present execution time breakdowns for superscalar processors.)

Busy-overhead: the time that the processor spends executing instructions that are not needed in the sequential program, but only in the parallel program. This corresponds directly to the extra work done in the parallel program.

Data-local: the time the processor is stalled waiting for a data reference it issued to be satisfied by the memory system on its own processing node; that is, waiting for a reference that does not require communication with other nodes.

Data-remote: the time it is stalled waiting for data to be communicated to or from another (remote) processing node, whether due to inherent or artifactual communication. This represents the cost of communication as seen by the processor.

Synchronization: the time it spends waiting for another process to signal the occurrence of an event that will allow it to proceed. This includes the load imbalance and serialization in the program, as well as the time spent actually executing synchronization operations and accessing synchronization variables. While it is waiting, the processor could be repeatedly polling a variable until that variable changes value—thus executing instructions—or it could be stalled, depending on how synchronization is implemented.[1]

The *synchronization*, *busy-overhead* and *data-remote* components are not found in a sequential program running on a uniprocessor, and are overheads introduced by parallelism. As we have seen, while inherent communication is mostly included in the *data-remote* component, some (usually very small) part of it might show up as *data-local* time as well. For example, data that is assigned to the local memory of a processor P might be updated by another processor Q, but asynchronously returned to P's memory (due to replacement from Q, say) before P references it. Finally, the *data-local* component is interesting, since it is a performance overhead in both the sequential and parallel cases. While the other overheads tend to increase with the number of processors for a fixed problem, this component may decrease. This is because the processor is responsible for only a portion of the overall calculation, so it may only access a fraction of the data that the sequential program does and thus obtain better local cache and memory behavior. If the *data-local* overhead reduces enough, it can give rise to *superlinear* speedups even for deter-

1. A parallel algorithm is deterministic if the result it yields for a given input data set are always the same independent of the number of processes used or the relative timings of events. More generally, we may consider whether all the intermediate calculations in the algorithm are deterministic. A non-deterministic algorithm is one in which the result and the work done by the algorithm to arrive at the result depend on the number of processes and relative event timing. An example is a parallel search through a graph, which stops as soon as any path taken through the graph finds a solution. Non-deterministic algorithms complicate our simple model of where time goes, since the parallel program may do less useful work than the sequential program to arrive at the answer. Such situations can lead to *superlinear* speedup, i.e., speedup greater than the factor by which the number of processors is increased. However, not all forms of non-determinism have such beneficial results. Deterministic programs can also lead to superlinear speedups due to greater memory system overheads in the sequential program than a in a parallel execution, as we shall see.

1. Synchronization introduces components of time that overlap with other categories. For example, the time to satisfy the processor's first access to the synchronization variable for the current synchronization event, or the time spent actually communicating the occurrence of the synchronization event, may be included either in synchronization time or in the relevant data access category. We include it in the latter. Also, if a processor executes instructions to poll a synchronization variable while waiting for an event to occur, that time may be defined as busy-overhead or as synchronization. We include it in synchronization time, since it is essentially load imbalance.

ministic parallel programs. Figure 3-13 summarizes the correspondences between parallelization

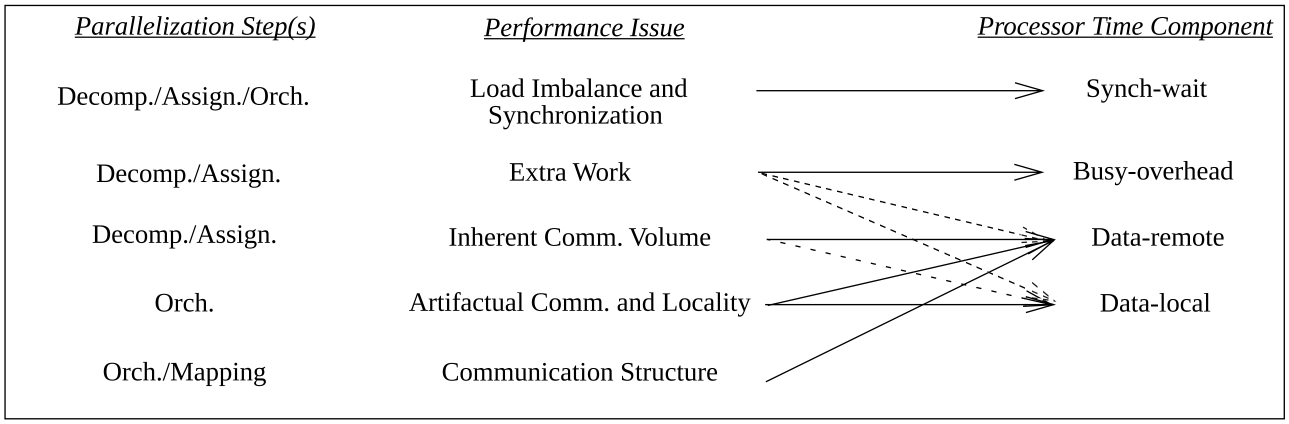


Figure 3-13 Mapping between parallelization issues and processor-centric components of execution time.

Bold lines depict direct relationships, while dotted lines depict side-effect contributions. On the left are shown the parallelization step in which the issues are mostly addressed.

issues, the steps in which they are mostly addressed, and processor-centric components of execution time.

Using these components, we may further refine our model of speedup for a fixed problem as follows, once again assuming a global synchronization at the end of the execution (otherwise we would take the maximum over processes in the denominator instead of taking the time profile of any single process):

$$\text{Speedup}_{prob}(p) = \frac{Busy(1) + Data_{\text{local}}(1)}{\text{Busy}_{useful}(p) + Data_{\text{local}}(p) + Synch(p) + Data_{\text{remote}}(p) + Busy_{overhead}(p)} \quad \text{(EQ 3.5)}$$

Our goal in addressing the performance issues has been to keep the terms in the denominator low and thus minimize the parallel execution time. As we have seen, both the programmer and the architecture have their roles to play. There is little the architecture can do to help if the program is poorly load balanced or if there is an inordinate amount of extra work. However, the architecture can reduce the incentive for creating such ill-behaved parallel programs by making communication and synchronization more efficient. The architecture can also reduce the artifactual communication incurred, provide convenient naming so that flexible assignment mechanisms can be easily employed, and make it possible to hide the cost of communication by overlapping it with useful computation.

3.6 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 finally ready to examine what we really wanted to do all along in software: to achieve good parallel performance on more realistic applications on real multiprocessors. In particular, we are now ready to return to the four application case studies that motivated us to study parallel software in the previous chapter, apply the four steps of the parallelization process to each case study, and at each step address the major performance issues that arise in it. In the process, we can understand and respond to the tradeoffs that arise among the different performance issues, as well as between performance and ease of programming. This will not only make our understanding of parallel software and tradeoffs more concrete, but 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 go forward through the rest of the book.

Parallel applications come in various shapes and sizes, with very different characteristics and very different tradeoffs among the major performance issues. Our four case studies provide an interesting though necessarily very restricted cross-section through the application space. In examining how to parallelize, and particularly orchestrate, them 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 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 particular machine of the chosen style: a 16-processor Silicon Graphics Origin2000. 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.

3.6.1 Ocean

Ocean, which simulates currents in an ocean basin, resembles many important applications in computational fluid dynamics. At each horizontal cross-section through the ocean basin, several different variables are modeled, including the current itself and the 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, non-border points that are actually computed in the equation solver is n-by-n). In all, about twenty-five 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 thirty three 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 equation solver kernel we have discussed so far. In the simple solver, each iteration is a sweep over 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 $\frac{n}{2} - \mathrm{by} - \frac{n}{2}$, $\frac{n}{4} - \mathrm{by} - \frac{n}{4}$, 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. To keep the computation deterministic and make it more efficient, a red-black ordering is used (see Section 2.4.2 on page 105).

Decomposition and Assignment

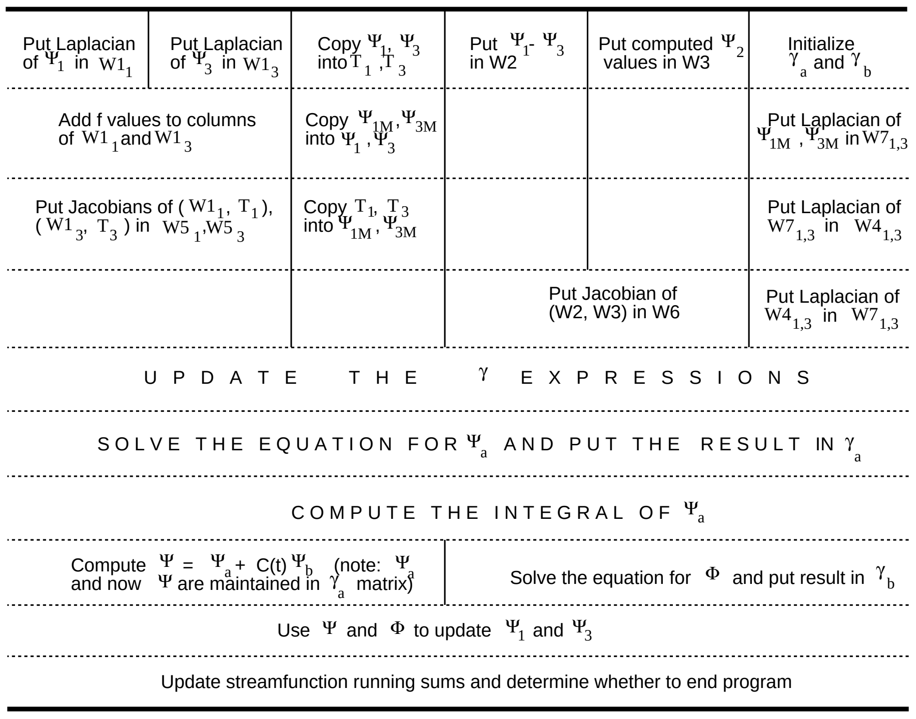
Ocean affords concurrency at two levels within a time-step: across some grid computations (function parallelism), and within a grid computation (data parallelism). Little or no concurrency is available across successive time-steps. Concurrency across grid computations is discovered by writing down which grids each computation reads and writes, and analyzing the dependences among them at this level. The 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.

There are several reasons why we choose to have all processes collaborate on each grid computation, rather than divide the processes among the available concurrent grid computations and use both levels of parallelism; i.e. why we concentrate on data parallelism. Combined data and function parallelism would increase the size of each process's partition of a grid, and hence reduce communication-to-computation ratio. However, the work associated with different grid computations is very varied and depends on problem size in different ways, which complicates load balanced assignment. 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 the same amount of work except at the borders, so we can statically assign grid points to processes using the data-parallel-only approach. Nonetheless, knowing which grid computations are independent is useful because it allows processes to avoid synchronizing between them.

The inherent communication issues are clearly very similar to those in the simple equation solver, so we use a block-structured domain decomposition of each grid. There is one complication—a tradeoff between locality and load balance related to the elements at the border of the entire grid. The internal *n*-by-*n* elements do similar work and are divided equally among all processes. Complete load balancing demands that border elements, which often do less work, also be divided equally among processors. However, communication and data locality suggest that border elements should be assigned to the processors that own the nearest internal elements. 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

[ARTIST: this figure is ugly in its current form. Anything you can do to beautify would be great. The idea is to have boxes that each represent a computation. Each row represents a computational phase. Boxes that are in the same horizontal row (phase) are independent of one another. The ones in the same vertical "column" are dependent, i.e. each depends on the one above it. I show these dependences by putting them in the same column, but the dependences could also be shown with arrows connecting the boxes (or ovals or whatever you choose to use). Showing the dependences among the computations is the main point of the figure.]


Put Laplacian of Ψ_1 in $W1_1$
Put Laplacian of Ψ_3 in $W1_3$
Copy Ψ_1, Ψ_3 into T_1, T_3
Put Ψ_1- Ψ_3 in W2
Put computed Ψ_2 values in W3
Initialize γ_a and γ_b
Add f values to columns of $W1_1$ and $W1_3$
Copy Ψ_{1M}, Ψ_{3M} into Ψ_1, Ψ_3
Put Laplacian of Ψ_{1M}, Ψ_{3M} in $W7_{1,3}$
Put Jacobians of ($W1_1$, T_1), ($W1_3$, T_3) in $W5_1$, $W5_3$
Copy T_1, T_3 into Ψ_{1M}, Ψ_{3M}
Put Laplacian of $W7_{1,3}$ in $W4_{1,3}$
Put Jacobian of (W2, W3) in W6
Put Laplacian of $W4_{1,3}$ in $W7_{1,3}$
UPDATE THE γ EXPRESSIONS
SOLVE THE EQUATION FOR Ψ_a AND PUT THE RESULT IN γ_a
COMPUTE THE INTEGRAL OF Ψ_a
Compute $\Psi = \Psi_a + C(t)\Psi_b$ (note: Ψ_a and now Ψ are maintained in γ_a matrix)
Solve the equation for Φ and put result in γ_b
Use Ψ and Φ to update Ψ_1 and Ψ_3
Update streamfunction running sums and determine whether to end program

Note: Horizontal lines represent synchronization points among all processes, and vertical lines spanning phases demarcate threads of dependence.

Figure 3-14 Ocean: The phases in a time-step and the dependences among grid computations.

Each box is a grid computation. Computations in the same row are independent, while those in the same column are dependent.

of grid points per processor decreases as we go to coarser levels of the hierarchy. At the highest *log p* of the *log n* possible levels, there will be less grid points than the *p* processors, so some processors will remain idle. Fortunately, relatively little 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 processor. This illustrates the importance of measuring speedups relative to the best sequential algorithm (here multigrid, for example): A classical, non-hierarchical iterative solver on the original grid would likely yield better *self-relative* speedups (relative to a single processor performing the same computation) than the 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"

Orchestration

Here we are mostly concerned with artifactual communication and data locality, and with the orchestration of 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 very similar to those discussed for the simple equation solver kernel in Section 3.4.1, and we use four-dimensional array data structures to represent the grids. 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) along column-oriented boundaries. There are two major differences between the simple solver and the complete Ocean application in issues related to spatial locality. The first is that because Ocean involves thirty-three different grid computations in every time-step, each involving one or more out of twenty-five different grids, we experience many conflict misses *across* grids. While we alleviate this by ensuring that the dimensions of the arrays that we allocate are not powers of two (even if the program uses power-of-two grids), it is difficult to lay different grids out relative to one another to minimizes conflict misses. The second difference has to do with the multigrid solver. The fact that a process's partition has fewer grid points at higher levels of the grid hierarchy makes it more difficult to allocate data appropriately at page granularity and reduces spatial locality, despite the use of four-dimensional arrays.

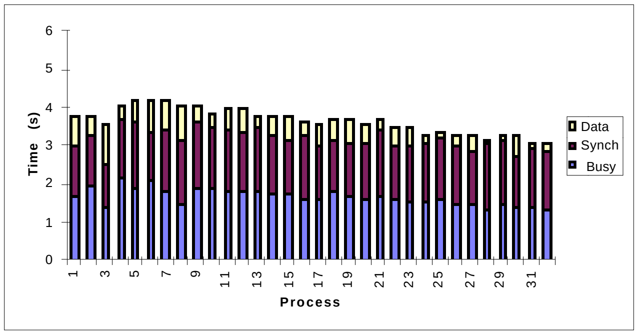
Working Sets and Temporal Locality: Ocean has a complicated working set hierarchy, with six working sets. These first three are due to the use of near-neighbor computations, including the multigrid solver, and are similar to those for the simple equation solver kernel. The first is captured when the cache is large enough to hold a few grid elements, so that an element that is accessed as the right neighbor for another element is reused to compute itself and also as the left neighbor for the next element. The second working set comprises a couple of 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 lead to a curve without sharp knees. The third 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 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 one to exploit 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 holds all the data that a process is assigned in every grid, so that all these data can be reused across times-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 grow linearly with the size of the data set *per process*. This growth rate is common in scientific applications that repeatedly stream through their data sets, so with large problems some important working sets do not fit in the local caches. Note that with proper data placement the working sets for a process consist mostly of local rather than communicated data. The little reuse that nonlocal data afford is captured by the first two working sets.

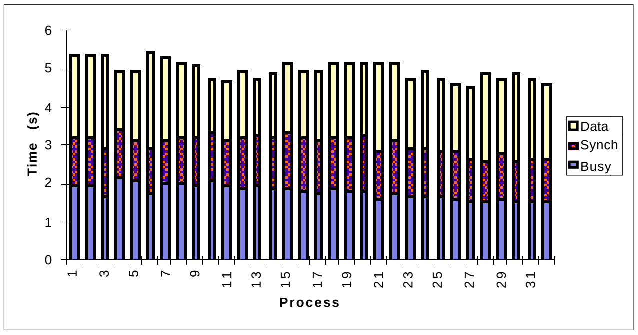
Synchronization: Ocean uses two types of synchronization. First, global barriers are used to synchronize all processes between computational phases (the horizontal lines in Figure 3-14), as well between iterations of the multigrid equation solver. Between several of the phases we could replace the barriers with finer-grained point-to-point synchronization at element level to obtain some overlap across phases; however, the overlap is likely to be too small to justify the overhead of many more synchronization operations and the programming complexity. Second, locks are used to provide mutual exclusion for global reductions. The work between synchronization points is very large, typically proportional to the size of a processor'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 adjacent to each other in the network topology. Our subgrid partitioning of two-dimensional grids clearly maps very well to a two-dimensional mesh network.



(a) Four-dimensional arrays



(a) Two-dimensional arrays

Figure 3-15 Execution time breakdowns for Ocean on a 32-processor Origin2000.

The size of each grid is 1030-by-1030, and the convergence tolerance is 10^{-3}. The use of four-dimensional arrays to represent the two-dimensional arrays to represent the two-dimensional grids 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 time fit very comfortably in the large 4MB 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.

In summary, Ocean is a good representative of many computational fluid dynamics computations that use regular grids. The computation to communication ratio is proportional to $\frac{n}{\sqrt{p}}$ 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 there is significant potential 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 1030-by-1030 grids on a 32-processor SGI Origin2000 machine. As in all our programs, mapping of processes to processors is not enforced by the program but is left to the system. This machine has very large per-processor second-level caches (4MB), so with four-dimensional arrays 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, while 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 place appropriately in main memory, many of these misses are not satisfied locally, leading to long latencies as well as contention.

3.6.2 Barnes-Hut

The galaxy simulation has far more irregular and dynamically changing behavior than Ocean. 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. Recall that the n-body problem is the problem of computing the influences that *n* bodies in a system exert on one another.

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

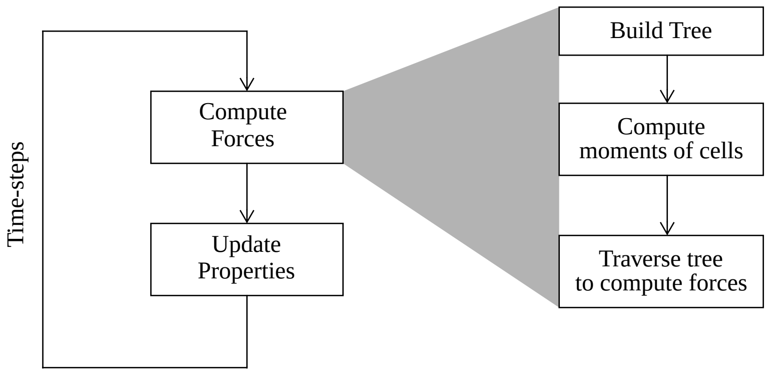


Figure 3-16 Flow of computation in the Barnes-Hut application.

that 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), then the effect of a large group of bodies may be approximated by a single equivalent body, if the 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.

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 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 cells and whose leaves are individual bodies.[1] 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 3-17 shows a small

1. 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.

two-dimensional example domain and the corresponding "quadtree" for simplicity. The positions of the bodies change across time-steps, so the tree has to be rebuilt every time-step. This results in the overall computational structure shown in the right hand side of Figure 3-16, with most of the time being spent in the force calculation phase.

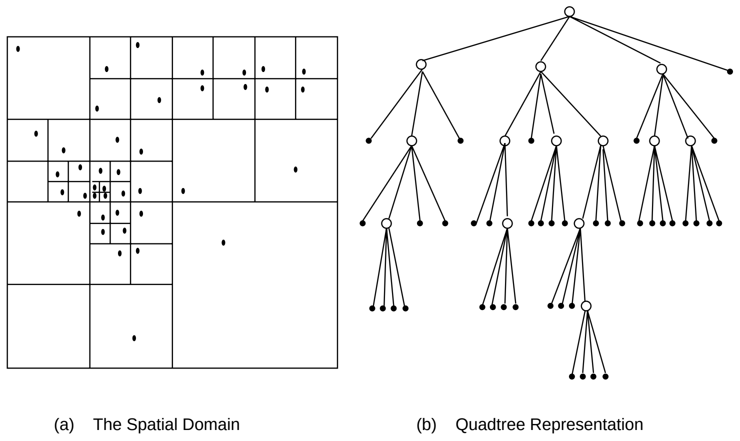


Figure 3-17 Barnes-Hut: A two-dimensional particle distribution and the corresponding quadtree.

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 computed. 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:

$$\frac{l}{d} < \theta, \qquad \textbf{(EQ 3.6)}$$

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 down those parts of the tree which represent 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 $O(\frac{1}{\theta^2} n \log n)$, since θ determines the number of tree cells touched at each level in a traversal.

Principal Data Structures

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 and acceleration, as well as its 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 an equal contiguous chunk of pointers in these arrays, which in every time-step are set to point to the bodies and cells that are assigned to it in that time-step. 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

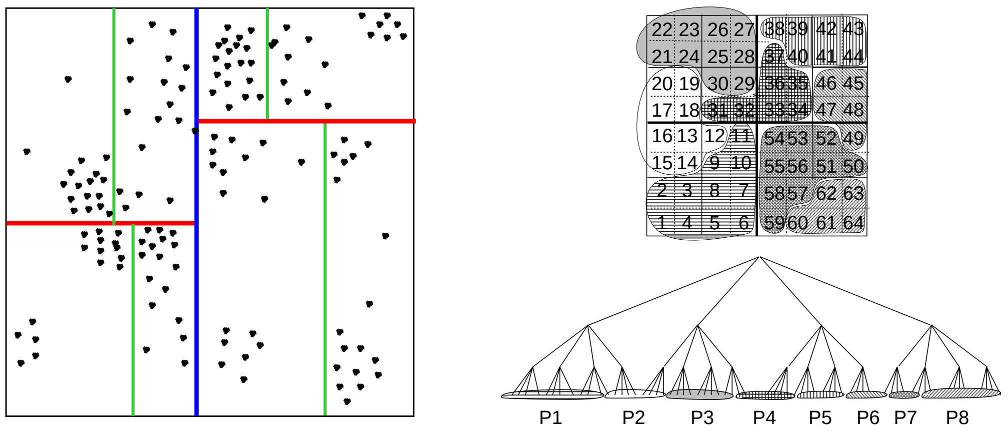
Each of the phases within a time-step is executed in parallel, with global barrier synchronization between phases. The natural unit of decomposition (task) in all phases is a body, except in computing the cell centers of mass, where it is a cell.

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 non-uniformity of the galaxy implies that the amount of work per body and the communication patterns are nonuniform, so a good assignment cannot be discovered by inspection. Second, the distribution of bodies changes across time-steps, which means that no static assignment is 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. And 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, while 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 since (a) the overhead of doing so (both in partitioning and in the loss of locality) outweighs the potential benefits, and (b) similar partitions are likely to work well for tree building and moment calculation (though not for the update phase).

As we have mentioned earlier in the chapter, we can use profiling-based semi-static partitioning in this application, taking advantage of the fact that although the particle distribution at the end of the simulation may be radically different from that at the beginning, it evolves slowly with time does not change very much 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. 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 very 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 we shall refer to it again in Section 3.7; the second because it 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 subspaces with equal cost, using the above load balancing measure, until there is one subspace per process (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 implement ORB. Details of using ORB for this application can be found in [Sal90].



(a) ORB (b) Costzones

Figure 3-18 Partitioning schemes for Barnes-Hut: ORB and Costzones.

ORB partitions space directly by recursive bisection, while costzones partitions the tree. (b) shows both the partitioning of the tree as well as how the resulting space is partitioned by costzones. Note that ORB leads to more regular (rectangular) partitions than costzones.

The second technique, called costzones, takes advantage of the fact 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 obtain the goal of partitioning space (see Figure 3-18(b)). Here is a high-level description. 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 1000 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 cost zone a body in the tree belongs to can be determined by the total cost of an inorder traversal of the tree up to that body. Processes traverse the tree in parallel, picking up the bodies that belong in their cost zone. Details can be found in [SH+95]. Costzones is much easier to implement than ORB. It also and yields better overall performance in a shared address space, mostly because the time spent in the partitioning phase itself is much smaller, illustrating 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 (see Section 3.7), data distribution to keep a process's assigned bodies and cells in its local main memory is not so easy as in Ocean.

First, data have to be redistributed dynamically as assignments change across time-steps, which is expensive. Second, the logical granularity of data (a particle/cell) is much smaller than the physical granularity of allocation (a page), and the fact that bodies/cells are spatially contiguous in the arrays does not mean they are contiguous in physical space or assigned to the same process. 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. Fortunately, there is enough temporal locality in the application that data distribution is not so important in a shared address space (again unlike Ocean). Also, the vast majority of the cache misses are to data in other processors' assigned partitions anyway, so data distribution itself wouldn't help make them 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 long cache blocks improve local access performance limited only by partition size, here multi-word 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 long transfer granularities might cause more fragmentation than useful prefetch, for the same reason that data distribution at page granularity is difficult: Unlike Ocean, locality of bodies/cells in the data structures does not match that in physical space on which assignment is based, so fetching data from more than one particle/cell upon a miss may be harmful rather than beneficial.

Working Sets and Temporal Locality: The first working set in this program contains the data used to compute forces between a single particle-particle or particle-cell pair. The interaction with the next particle or cell in the traversal will reuse these 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 particle. Because of the way partitioning is done, the traversal to compute the forces on the next particle will reuse most of these data. As we go from particle to particle, the composition of this working set changes slowly. However, 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 these data are allocated nonlocally. Thus, it is the temporal locality exploited on shared (both local and nonlocal) data that is critical to the performance of the application, unlike Ocean where it is data distribution.

By the same reasoning that the complexity of the algorithm is $O(\frac{1}{\theta^2} n \log n)$, the expected size of this working set is proportional to $O(\frac{1}{\theta^2} \log n)$, 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 fits comfortably in modern second-level caches, we do not replicate data in main memory. In Ocean there were important working sets that grew linearly with the data set size, and we did not always expect them to fit in the cache; however, even there we did not need replication in main memory since if data were distributed appropriately then the data in these working sets were local.

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 much more difficult to replace the barriers by point-to-point body or cell level synchronization. The small number of barriers used in a time-step is independent of problem size or number of processors.

There is no need for synchronization within the force computation phase itself. While communication and sharing patterns in the application are irregular, they are phase-structured. That is, while a process reads particle and cell data from many other processes in the force calculation phase, the fields of a particle 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. 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 particle 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 moment calculation phase 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 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 $O(\frac{n}{p})$, 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

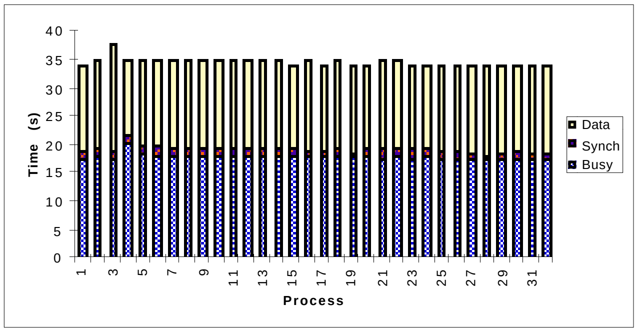
The irregular nature makes this application more difficult to map perfectly for network locality in common networks such as meshes. The ORB partitioning scheme maps very naturally to a hypercube topology (discussed in Chapter 10), but not so well to a mesh or other less richly interconnected network. 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 even in such a network.

In summary, the Barnes-Hut application has 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 it are not obvious by inspection of the code, 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.

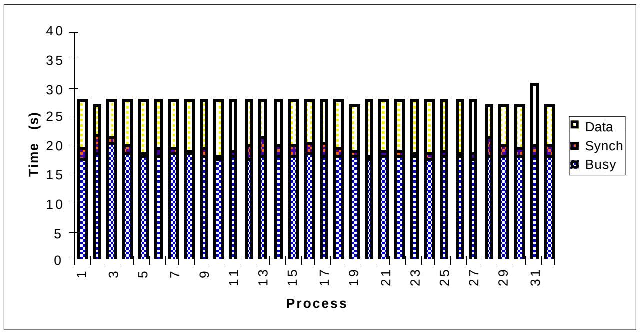
Figure 3-19 shows the breakdowns of execution time for this application. Load balance is quite good with a static partitioning of the array of bodies to processors, precisely because there is little relationship between their location in the array and in physical space. However, the data access cost is high, since there is a lot of inherent and artifactual communication. Semi-static, costzones partitioning reduces this data access overhead substantially without compromising load balance.

3.6.3 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 traced as they bounce around to compute a color and opac-



(a) Static assignment of bodies



(a) Semi-static, costzones assignment

Figure 3-19 Execution time breakdown for Barnes-Hut with 512K 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. This is why data wait time is much smaller in the semi-static 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.

ity 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 octree used by the Barnes-Hut application. The root of the tree represents the entire space enclosing the scene, and the leaves hold a linked list of the object primitives that fall in them (the maximum number of primitives per leaf is also defined by the user). 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.

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 either 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 tradeoffs between execution time and image quality, and many algorithmic optimizations have been developed to improve performance without compromising image quality much.

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 interactions for rays 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 alternate, *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 follows a ray in its path through the entire scene, computing the interactions of the entire ray tree which that ray generates. The unit of decomposition here is a primary ray. It 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 that are in its subspace and the rays that enter that subspace. However, the ray-oriented approach is much easier to implement with low overhead, particularly starting from a sequential program, since rays can be processed independently without synchronization and the scene data are read-only. This program therefore uses a ray-oriented approach. The degree of concurrency for a *n*-by-*n* plane of pixels is $O(n^2)$, and is usually ample.

Unfortunately, a static subblock 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 queueing system (one queue per processor) with task stealing for load balancing.

Consider communication. Since the scene data are read-only, there is no inherent communication on these data. If we replicated the entire scene on every node, there would be no communication except due to task stealing. However this approach does not allow us to render a scene larger than what fits in a single processor's memory. Other than task stealing, communication is generated because only $1/p$ of the scene is allocated locally and 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 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 be reflected in similar ways. This suggests that we should use domain decomposition on the image plane to assign pixels to task queues initially. Since the adjacency or spatial coherence of rays works in all directions in the image plane, block-oriented 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 sized 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 that is assigned that block. A processor ray traces the tiles in its block in scan-line order. When it is done 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 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.

Orchestration

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

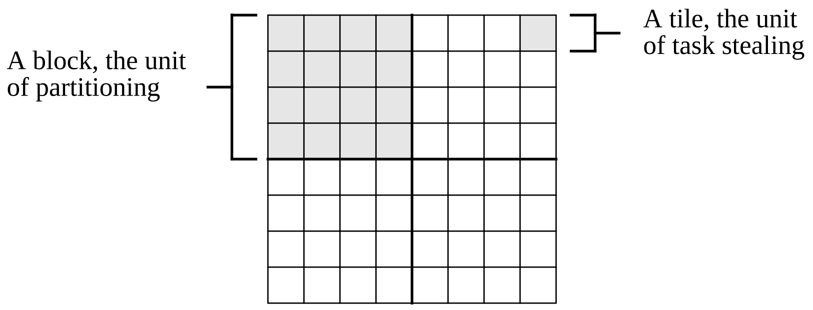

A block, the unit of partitioning
A tile, the unit of task stealing

Figure 3-20 Image plane partitioning in Raytrace for four processors.

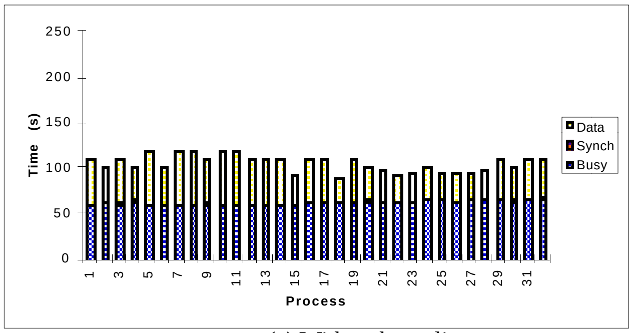
Spatial Locality: Most of the shared data accesses are to the scene data. However, because of changing viewpoints and the fact that rays bounce about unpredictably, it is impossible to divide the scene into parts that are each accessed only (or even dominantly) by a single process. Also, the scene data structures are naturally small 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 contention. Image data are small, and we try to allocate the few pages they fall on in different memories as well. The sub-block partitioning described above preserves spatial locality at cache block granularity in the image plane quite well, though it can lead to some false sharing 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 as well. 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 a nonlocally allocated datum would cause communication. With finite replication capacity, on the other hand, data may be replaced and 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 reference 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 and replication in main memory has a cost, so it is unclear that the benefits outweigh the overheads.

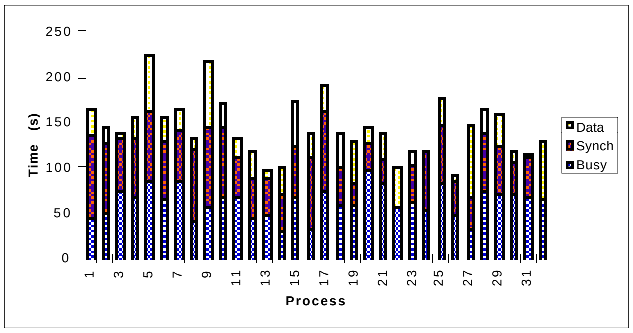
Synchronization and Granularity: There is only a single barrier after an entire scene is rendered and before it is displayed. Locks are used to protect task queues for task stealing, and also for some global variables that track statistics for the program. The work between synchronization points is the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

Mapping

Since Raytrace has very unpredictable access and communication patterns to scene data, the communication is all but impossible to map effectively in this shared address space version. The fact that the initial assignment of rays partitions the image into a two-dimensional grid of blocks, it would be natural to map to a two-dimensional mesh network but the effect is not likely to be large.



(a) With task stealing



(a) Without task stealing

Figure 3-21 Execution time breakdowns 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) in this highly unpredictable application.

In summary, this application tends to have large working sets and relatively poor spatial locality, but a low inherent communication to computation ratio. Figure 3-21 shows the breakdown of execution time for the balls data set, illustrating the importance of task stealing in reducing load imbalance. The extra communication and synchronization incurred as a result is well worthwhile.

3.6.4 Data Mining

A key difference in the data mining application from the previous ones is that the data being accessed and manipulated typically reside on disk rather than in memory. It is very important to reduce the number of disk accesses, since their cost is very high, and also to reduce the contention for a disk controller by different processors.

Recall the basic insight used in association mining from Section 2.2.4: If an itemset of size *k* is large, then all subsets of that itemset must also be large. For illustration, consider a database in which there are 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 2. 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; i.e. taking pairs of itemsets in L_2 that share a common first item (say AB and AC), and combining them into a lexicographically sorted 3-itemset (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 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 item 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 the example above). 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.

Sequential Algorithm

A simple sequential method for association mining is to first traverse the dataset 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 dataset again to find which entries of C_2 are large and should be in L_2. From L_2, we can construct C_3, and traverse the dataset to determine L_3, and so on until we have found L_k. While this method is simple, it requires reading all transactions in the database from disk *k* times, which is expensive.

The key goals in a sequential algorithm are 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 in determining the counts of itemsets in a candidate list C_k (to determine which itemsets should be in L_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 the data are 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, T1, T2, T3, ...}, where IS_x is an itemset and T1, T2 etc. are transactions that contain that itemset. That is, there is a database record per itemset rather than per transaction. If the large itemsets of size k-1 (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. If each itemset has its list of transactions attached to it, as in the above representation, then the size of each resulting itemset in C_k can be computed at the same time as identifying the C_k itemset itself from a pair of L_{k-1} itemsets, by computing the intersection of the transactions in their lists.

For example, suppose {AB, 1, 3, 5, 8, 9} and {AC, 2, 3, 4, 8, 10} are large 2-itemsets in the same 1-equivalence class (they each start with A), then the list of transactions that contain itemset ABC is {3, 8}, so the occurrence count of itemset ABC is two. What this means is that once the database is transposed and the 1-equivalence classes identified, the rest of the computation for a single 1-equivalence class can be done to completion (i.e. all large k-itemsets found) before considering any data from other 1-equivalence classes. If a 1-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.

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 requires 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 1-equivalence class is independent of the computation on any other, we can simply divide up the 1-equivalence classes among processes which can thereafter proceed independently, 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 there is no need for remote disk access after this point either. As in the sequential algorithm, since each process can complete the work on one of its assigned equivalence classes before proceeding to the next one, hopefully each item from the local database should be read only once. The issue 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 k-itemsets, 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, one may have to resort to dynamic tasking and task stealing, which can compromise much of the simplicity of this method (e.g. that once processes are assigned their initial equivalence classes they do not have to communicate, synchronize, or perform remote disk access).

The first step in this approach, of course, is to compute the 1-equivalence classes and the large 2-itemsets in them, as a starting point for the parallel assignment. To compute the large 2-itemsets, we are better off using the original form of the database rather than the transformed form, so we do not transform the database yet (see Exercise 3.12). 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 array, with the indices being items). It is easy to compute 2-itemset counts directly, rather than first make a pass to compute 1-itemset counts, construct the list of large 1-itemsets, and then make another pass to compute 2-itemset counts. The local counts are then merged, involving interprocess communication, and the large 2-itemsets determined from the resulting global counts. These 2-itemsets are then partitioned into 1-equivalence classes, and assigned to processes as described above.

The next step is to transform the database from the original {T_x, A, B, D, ...} organization by transaction to the {IS_x, T1, T2, T3, ...} organization by itemset, where the IS_x are initially the 2-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 2-itemsets from its local portion of the database. Then, in the communication step a process (at least conceptually) "sends" the lists for those 2-itemsets whose 1-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 k-itemsets step by step for each of its equivalence classes, without any communication, synchronization or remote disk access. The communication step of the transformation phase is usually the most expensive step in the algorithm. Finally, the results for the large k-itemsets are available from the different processes.

Orchestration

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

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

Temporal Locality: 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 fit 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 we are exploiting temporal locality in main memory rather than in the cache, although the techniques and goals are similar.

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 2-itemsets), and a barrier after this to begin the transformation phase. The reduction is required only for 2-itemsets, since thereafter every process proceeds independently to compute the large k-itemsets in its assigned equivalence classes. Further synchronization may be needed if dynamic task management is used for load balancing.

This concludes our in-depth discussion of how the four case studies might actually be parallelized. At least in the first three, we assumed a coherent shared address space programming model in the discussion. Let us now see how the characteristics of these case studies and other applications influence the tradeoffs between the major programming models for multiprocessors.

3.7 Implications for Programming Models

We have seen in 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, and that the positions taken on these influence (and are influenced by) performance characteristics such as latency and bandwidth. At that stage, we could only speak about these issues in the abstract, and could not appreciate the interactions with applications and the implications for which programming models are preferable under what circumstances. Now that we have an in-depth understanding of several interesting parallel applications, and we understand the performance issues in orchestration, we are ready to compare the programming models in light of application and performance characteristics.

We will use the application case studies to illustrate the issues. For a shared address space, we assume that loads and stores to shared data are the only communication mechanisms exported to the user, and we call this a load-store shared address space. Of course, in practice there is nothing to stop 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 layers (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 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. For both the shared address space and message passing models, we assume that the programming model is supported efficiently by the communication abstraction and the hardware-software interface.

As programmers, the programming model is 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. The major aspects that distinguish the two programming models are functional aspects like *naming*, *replication* and *synchronization*, organizational aspects like the *granularity* at which communication is performed, and performance aspects such as end-point *overhead* of a communication operation. Other performance aspects such as latency and bandwidth depend largely on the network and network interface used, and can be assumed to be equivalent. In addition, there are differences in *hardware overhead and complexity* required to support the abstractions efficiently, and in the ease with which they allow us to reason about or *predict performance*. Let us examine each of these aspects. The first three point to advantages of a load/store shared address space, while the others favor explicit message passing.

Naming

We have already seen that a shared address space makes the naming of logically shared data much easier for the programmer, since the naming model is similar to that on a uniprocessor. Explicit messages are not necessary, and a process need not name other processes or know which processing node currently owns the data. Having to know or determine which process's address space data reside in and transfer data in explicit messages is not difficult in applications with regular, statically predictable communication needs, such as the equation solver kernel and Ocean. But it can be quite difficult, both algorithmically and for programming, in applications with irregular, unpredictable data needs. An example is Barnes-Hut, in which the parts of the tree that a process needs to compute forces on its bodies are not statically predictable, and the ownership of bodies and tree cells changes with time as the galaxy evolves, so knowing which processes to obtain data from is not easy. Raytrace is another example, in which rays shot by a process bounce unpredictably around scene data that are distributed among processors, so it is difficult to determine who owns the next set of data needed. These difficulties can of course be overcome in both cases, but this requires either changing the algorithm substantially from the uniprocessor version (for example, adding an extra time-step to compute who needs which data and transferring those data to them in Barnes-Hut [Sal90], or using a scene-oriented rather than a ray-oriented approach in Raytrace, as discussed in Section 3.6.3) or emulating an application-specific shared address space in software by hashing bodies, cells or scene data to processing nodes. We will discuss these solutions further in Chapter 7 when we discuss the implications of message passing systems for parallel software.

Replication

Several issues distinguish how replication of nonlocal data is managed on different system: (i) *who* is responsible for doing the replication, i.e. for making local copies of the data? (ii) *where* in the local memory hierarchy is the replication done? (iii) at *what granularity* are data allocated in the replication store? (iv) how are the values of replicated data kept *coherent*? (v) how is the *replacement* of replicated data managed?

With the separate virtual address spaces of the message-passing abstraction, the only way to replicate communicated data is to copy the data into a process's private address space explicitly in the application program. The replicated data are explicitly renamed in the new process's private address space, so the virtual and physical addresses may be different for the two processes and their copies have nothing to do with each other as far as the system is concerned. Organizationally, data are always replicated in main memory first (when the copies are allocated in the address space), and only data from the local main memory enter the processor cache. The granularity of allocation in the local memory is variable, and depends entirely on the user. Ensuring that the values of replicated data are kept up to date (coherent) must be done by the program through explicit messages. We shall discuss replacement shortly.

We mentioned in Chapter 1 that in a shared address space, since nonlocal data are accessed through ordinary processor loads and stores and communication is implicit, there are opportunities for the system to replicate data transparently to the user—without user intervention or explicit renaming of data—just as caches do in uniprocessors. This opens up a wide range of possibilities. For example, in the shared physical address space system we assume, nonlocal data enter the processor's cache subsystem upon reference by a load or store instruction, without being replicated in main memory. Replication happens very close to the processor and at the relatively fine granularity of cache blocks, and data are kept coherent by hardware. Other systems may replicate data transparently in main memory—either at cache block granularity through additional hardware support or at page or object granularity through system software—and may preserve coherence through a variety of methods and granularities that we shall discuss in Chapter 9. Still other systems may choose not to support transparent replication and/or coherence, leaving them to the user.

Finally, let us examine the replacement of locally replicated data due to finite capacity. How replacement is managed has implications for the amount of communicated data that needs to be replicated at a level of the memory hierarchy. For example, hardware caches manage replacement automatically and dynamically with every reference and at a fine spatial granularity, so that the cache needs to be only as large as the active working set of the workload (replication at coarser spatial granularities may cause fragmentation and hence increase the replication capacity needed). When replication is managed by the user program, as in message passing, it is difficult to manage replacement this dynamically. It can of course be done, for example by maintaining a cache data structure in local memory and using it to emulate a hardware cache for communicated data. However, this complicates programming and is expensive: The software cache must be looked up in software on every reference to see if a valid copy already exists there; if it does not, the address must be checked to see whether the datum is locally allocated or a message should be generated.

Typically, message-passing programs manage replacement less dynamically. Local copies of communicated data are allowed to accumulate in local memory, and are flushed out explicitly at certain points in the program, typically when it can be determined that they are not needed for some time. This can require substantially more memory overhead for replication in some applications such as Barnes-Hut and Raytrace. For read-only data, like the scene in Raytrace many message passing programs simply replicate the entire data set on every processing node, solving the naming problem and almost eliminating communication, but also eliminating the ability to run larger problems on larger systems. In the Barnes-Hut application, while one approach emulates a shared address space and also a cache for replication—"flushing" the cache at phase boundaries—in the other approach a process first replicates locally all the data it needs to compute forces on all its assigned bodies, and only then begins to compute forces. While this means there is no communication during the force calculation phase, the amount of data replicated in main memory is much larger than the process's assigned partition, and certainly much larger than the active working set which is the data needed to compute forces on only one particle (Section 3.6.2). This active working set in a shared address space typically fits in the processor cache, so there is not need for replication in main memory at all. In message passing, the large amount of replication limits the scalability of the approach.

Overhead and Granularity of Communication

The overheads of initiating and receiving communication are greatly influenced by the extent to which the necessary tasks can be performed by hardware rather than being delegated to software, particularly the operating system. As we saw in Chapter 1, in a shared physical address space the underlying uniprocessor hardware mechanisms suffice for address translation and protection, since the shared address space is simply a large flat address space. Simply doing address translation in hardware as opposed to doing it in software for remote references was found to improve Barnes-Hut performance by about 20% in one set of experiments [ScL94]. The other major task is buffer management: incoming and outgoing communications need to be temporarily buffered in the network interface, to allow multiple communications to be in progress simultaneously and to stage data through a communication pipeline. Communication at the fixed granularity of cache blocks makes it easy to manage buffers very efficiently in hardware. These factors combine to keep the overhead of communicating each cache block quite low (a few cycles to a few tens of cycles, depending on the implementation and integration of the communication assist).

In message-passing systems, local references are just loads and stores and thus incur no more overhead than on a uniprocessor. Communication messages, as we know, are much more flexible. They are typically of a variety of possible types (see Section 2.4.6 and the tag matching discussed in Chapter 1), of arbitrary lengths, and between arbitrary address spaces. The variety of types requires software overhead to decode the type of message and execute the corresponding handler routine at the sending or receiving end. The flexible message length, together with the use of asynchronous and nonblocking messages, complicates buffer management, so that the operating system must often be invoked to temporarily store messages. Finally, sending explicit messages between arbitrary address spaces requires that the operating system on a node intervene to provide protection. The software overhead needed to handle buffer management and protection can be substantial, particularly when the operating system must be invoked. A lot of recent design effort has focused on streamlined network interfaces and message-passing mechanisms, which have significantly reduced per-message overheads. These will be discussed in Chapter 7. However, the issues of flexibility and protection remain, and the overheads are likely to remain several times as large as those of load-store shared address space interfaces. Thus, explicit message-passing is not likely to sustain as fine a granularity of communication as a hardware-supported shared address space.

These three issues—naming, replication, and communication overhead—have pointed to the advantages of an efficiently supported shared address space from the perspective of creating a parallel program: The burdens of naming and often replication/coherence are on the system rather than the program, and communication need not be structured in large messages to amortize overhead but can be done naturally through loads and stores. The next four issues, however, indicate advantages of explicit communication.

Block Data Transfer

Implicit communication through loads and stores in a cache-coherent shared address space typically causes a message to be generated per reference that requires communication. The communication is usually initiated by the process that needs the data, and we call it receiver-initiated. Each communication brings in a fixed, usually small amount of data: a cache block in our hardware cache-coherent system. While the hardware support provides efficient fine-grained communication, communicating one cache block at a time is not the most efficient way to communicate a large chunk of data from one processor to another. We would rather amortize the overhead and latency by communicating the data in a single message or a group of large messages. Explicit communication, as in message passing, allows greater flexibility in choosing the sizes of messages and also in choosing whether communication is receiver-initiated or sender-initiated. Explicit communication can be added to a shared address space naming model, and it is also possible for the system to make communication coarser-grained transparently underneath a load-store programming model in some cases of predictable communication. However, the natural communication structure promoted by a shared address space is fine-grained and usually receiver-initiated. The advantages of block transfer are somewhat complicated by the availability of alternative latency tolerance techniques, as we shall see in Chapter 11, but it clearly does have advantages.

Synchronization

The fact that synchronization can be contained in the (explicit) communication itself in message passing, while it is usually explicit and separate from data communication in a shared address space is an advantage of the former model. However, the advantage becomes less significant when asynchronous message passing is used and separate synchronization must be employed to preserve correctness.

Hardware Cost and Design Complexity

The hardware cost and design time required to efficiently support the desirable features of a shared address space above are greater those required to support a message-passing abstraction. Since all transactions on the memory bus must be observed to determine when nonlocal cache misses occur, at least some functionality of the communication assist be integrated quite closely into the processing node. A system with transparent replication and coherence in hardware caches requires further hardware support and the implementation of fairly complex coherence protocols. On the other hand, in the message-passing abstraction the assist does not need to see memory references, and can be less closely integrated on the I/O bus. The actual tradeoffs in cost and complexity for supporting the different abstractions will become clear in Chapters 5, 7 and 8.

Cost and complexity, however, are more complicated issues than simply assist hardware cost and design time. For example, if the amount of replication needed in message-passing programs is indeed larger than that needed in a cache-coherent shared address space (due to differences in how replacement is managed, as discussed earlier, or due to replication of the operating system), then the memory required for this replication should be compared to the hardware cost of supporting a shared address space. The same goes for the recurring cost of developing effective programs on a machine. Cost and price are also determined largely by volume in practice.

Performance Model

In designing parallel programs for an architecture, we would like to have at least a rough performance model that we can use to predict whether one implementation of a program will be better than another and to guide the structure of communication. There are two aspects to a performance model: modeling the cost of primitive events of different types—e.g. communication messages—and modeling the occurrence of these events in a parallel program (e.g. how often they occur, in how bursty a manner, etc.). The former is usually not very difficult, and we have seen a simple model of communication cost in this chapter. The latter can be quite difficult, especially when the programs are complex and irregular, and it is this that distinguishes the performance modeling ease of a shared address space from that of message passing. This aspect is significantly easier when the important events are explicitly identified than when they are not. Thus, the message passing abstraction has a reasonably good performance model, since all communication is explicit. The performance guidelines for a programmer are at least clear: *messages are expensive; send them infrequently.* In a shared address space, particularly a cache-coherent one, performance modeling is complicated by the same property that makes developing a program easier: Naming, replication and coherence are all implicit, so it is difficult to determine how much communication occurs and when. Artifactual communication is also implicit and is particularly difficult to predict (consider cache mapping conflicts that generate communication!), so the resulting programming guideline is much more vague: try to exploit temporal and spatial locality and use data layout when necessary to keep communication levels low. The problem is similar to how implicit caching can make performance difficult to predict even on a uniprocessor, thus complicating the use of the simple *von Neumann* model of a computer which assumes that all memory references have equal cost. However, it is of much greater magnitude here since the cost of communication is much greater than that of local memory access on a uniprocessor.

In summary, the major potential advantages of implicit communication in the shared address space abstraction are programming ease and performance in the presence of fine-grained data sharing. The major potential advantages of explicit communication, as in message passing, are the benefits of block data transfer, the fact that synchronization is subsumed in message passing, the better performance guidelines and prediction ability, and the ease of building machines.

Given these tradeoffs, the questions that an architect has to answer are:

Is it worthwhile to provide hardware support for a shared address space (i.e. transparent naming), or is it easy enough to manage all communication explicitly?

If a shared address space is worthwhile, is it also worthwhile to provide hardware support for transparent replication and coherence?

If the answer to either of the above questions is yes, then is the implicit communication enough or should there also be support for explicit message passing among processing nodes that can be used when desired (this raises a host of questions about how the two abstractions should be integrated), which will be discussed further in Chapter 11.

The answers to these questions depend both on application characteristics and on cost. Affirmative answers to any of the questions naturally lead to other questions regarding how efficiently the feature should be supported, which raise other sets of cost, performance and programming tradeoffs that will become clearer as we proceed through the book. Experience shows that as applications become more complex, the usefulness of transparent naming and replication increases, which argues for supporting a shared address space abstraction. However, with communication naturally being fine-grained, and large granularities of communication and coherence causing performance problems, supporting a shared address space effectively requires an aggressive communication architecture.

3.8 Concluding Remarks

The characteristics of parallel programs have important implications for the design of multiprocessor architectures. Certain key observations about program behavior led to some of the most important advances in uniprocessor computing: The recognition of temporal and spatial locality in program access patterns led to the design of caches, and an analysis of instruction usage led to reduced instruction set computing. In multiprocessors, the performance penalties for mismatches between application requirements and what the architecture provides are much larger, so it is all the more important that we understand the parallel programs and other workloads that are going to run on these machines. This chapter, together with the previous one, has focussed on providing this understanding.

Historically, many different parallel architectural genres led to many different programming styles and very little portability. Today, the architectural convergence has led to common ground for the development of portable software environments and programming languages. The way we think about the parallelization process and the key performance issues is largely similar in both the shared address space and message passing programming models, although the specific granularities, performance characteristics and orchestration techniques are different. While we can analyze the tradeoffs between the shared address space and message passing programming models, both are flourishing in different portions of the architectural design space.

Another effect of architectural convergence has been a clearer articulation of the performance issues against which software must be designed. Historically, the major focus of theoretical parallel algorithm development has been the PRAM model discussed in Section 3.2.4, which ignores data access and communication cost and considers only load balance and extra work (some variants of the PRAM model some serialization when different processors try to access the same data word). This is very useful in understanding the inherent concurrency in an application, which is the first conceptual step; however, it does not take important realities of modern systems into account, such as the fact that data access and communication costs can easily dominate execution time. Historically, communication has been treated separately, and the major focus in its treatment has been mapping the communication to different network topologies. With a clearer understanding of the importance of communication and the important costs in a communication transaction on modern machines, two things have happened. First, models that help analyze communication cost and structure communication have been developed, such as the bulk synchronous programming (BSP) model [Val90] and the LogP model [CKP+93], with the hope of replacing the PRAM as the de facto model used for parallel algorithm analysis. These models strive to expose the important costs associated with a communication event—such as latency, bandwidth, overhead—as we have done in this chapter, allowing an algorithm designer to factor them into the comparative analysis of parallel algorithms. The BSP model also provides a framework that can be used to reason about communication and parallel performance. Second, the emphasis in modeling communication cost has shifted to the cost of communication at the end points, so the number of messages and contention at the end points have become more important than mapping to network topologies. In fact, both the BSP and LogP models ignore network topology completely, modeling network latency as a fixed parameter!

Models such as BSP and LogP are important steps toward a realistic architectural model against which to design and analyze parallel algorithms. By changing the values of the key parameters in these models, we may be able to determine how an algorithm would perform across a range of architectures, and how it might be best structured for different architectures or for portable performance. However, much more difficult than modeling the architecture as a set of parameters is modeling the behavior of the algorithm or application, which is the other side of the modeling equation [SRG94]: What is the communication to computation ratio for irregular applications, how does it change with replication capacity, how do the access patterns interact with the granularities of the extended memory hierarchy, how bursty is the communication and how can this be incorporated into the performance model. Modeling techniques that can capture these characteristics for realistic applications and integrate them into the use of BSP or LogP have yet to be developed.

This chapter has discussed some of the key performance properties of parallel programs and their interactions with basic provisions of a multiprocessor's memory and communications architecture. These properties include load balance; the communication to computation ratio; aspects of orchestrating communication that affect communication cost; data locality and its interactions with replication capacity and the granularities of allocation, transfer and perhaps coherence to generate artifactual communication; and the implications for communication abstractions that a machine must support. We have seen that the performance issues trade off with one another, and that the art of producing a good parallel program is to obtain the right compromise between the conflicting demands. Performance enhancement techniques can take considerable programming effort, depending on both the application and the system, and the extent and manner in which different techniques are incorporated can greatly affect the characteristics of the workload presented to the architecture. Programming for performance is also a process of successive refinement, where decisions made in the early steps may have to be revisited based on system or program characteristics discovered in later steps. We have examined the four application case studies that were introduced in Chapter 2 in depth, and seen how these issues play out in them. We shall encounter several of these performance issues again in more detail as we consider architectural design options, tradeoffs and evaluation in the rest of the book. However, with the knowledge of parallel programs that we have developed, we are now ready to understand how to use the programs as workloads to evaluate parallel architectures and tradeoffs.

3.9 References

[AiN88] Alexander Aiken and Alexandru Nicolau. Optimal Loop Parallelization. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 308-317, 1988.

[AC+95] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg and Katherine Yelick. Empirical Evaluation of the Cray-T3D: A Compiler Perspective. Proceedings of the Twenty-second International Symposium on Computer Architecture, pp. 320-331, June 1995.

[Bu+92] Burkhardt, H. et al, Overview of the KSR-1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[CM88] Chandy, K.M. and Misra, J. Parallel Program Design: A Foundation. Addison Wesley, 1988.

[CKP+93] David Culler et al. LogP: Toward a realistic model of parallel computation. In *Proceedings of the Principles and Practice of Parallel Processing*, pages 1-12, 1993.

[Da+87] Dally, W.J. et al. The J-Machine: A Fine-grained Concurrent Computer. Proceedings of the IFIPS

World Computer Congress, pp. 1147-1153, 1989.

[Den68] Denning, P.J. The Working Set Model for Program Behavior. Communications of the ACM, vol. 11, no. 5, May 1968, pp. 323-333.

[DS68] Dijkstra, E.W. and Sholten, C.S. Termination Detection for Diffusing Computations. Information Processing Letters 1, pp. 1-4.

[FoW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In Proceedings of the Tenth ACM Symposium on Theory of Computing, May 1978.

[GP90] Green, S.A. and Paddon, D.J. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, vol. 6, 1990, pp. 62-73.

[Hil85] Hillis, W.D. The Connection Machine. MIT Press, 1985.

[HiS86] Hillis, W.D. and Steele, G.L. Data Parallel Algorithms. Communications of the ACM, 29(12), pp. 1170-1183, December 1986.

[HoC92] Hockney, R. W., and <<xxx>>. Comparison of Communications on the Intel iPSC/860 and Touchstone Delta. Parallel Computing, 18, pp. 1067-1072, 1992.

[KG+94] V. Kumar, A.Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Lei+92] Charles E. Leiserson, Z.S. Abuhamdeh, D. C. Douglas, C.R. Feynmann, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M. St. Pierre, D.S. Wells, M.C. Wong, S-W Yang and R. Zak. The Network Architecture of the Connection Machine CM-5. In Proceedings of the Symposium on Parallel Algorithms and Architectures, pp. 272-285, June 1992.

[LLJ+93] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[LO+87] Lusk, E.W., Overbeek, R. et al. Portable Programs for Parallel Processors. Holt, Rinehart and Winston, Inc. 1987.

[NWD93] Noakes, M.D., Wallach, D.A., and Dally, W.J. The J-Machine Multicomputer: An Architectural Evaluation. Proceedings of the Twentieth International Symposium on Computer Architecture, May 1993, pp. 224-235.

[Pie88] Pierce, Paul. The NX/2 Operating System. Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pp. 384--390, January, 1988.

[Sal90] Salmon, J. Parallel Hierarchical N-body Methods. Ph.D. Thesis, California Institute of Technology, 1990.

[WaS93] Warren, Michael S. and Salmon, John K. A Parallel Hashed Oct-Tree N-Body Algorithm. Proceedings of Supercomputing'93, IEEE Computer Society, pp. 12-21, 1993.

[SWW94] Salmon, J.K., Warren, M.S. and Winckelmans, G.S. Fast parallel treecodes for gravitational and fluid dynamical N-body problems. Intl. Journal of Supercomputer Applications, vol. 8, pp. 129 - 142, 1994.

[ScL94] D. J. Scales and M. S. Lam. Proceedings of the First Symposium on Operating System Design and Implementation, November, 1994.

[SGL94] Singh, J.P., Gupta, A. and Levoy, M. Parallel Visualization Algorithms: Performance and Architectural Implications. IEEE Computer, vol. 27, no. 6, June 1994.

[SH+95] Singh, J.P., Holt, C., Totsuka, T., Gupta, A. and Hennessy, J.L. Load balancing and data locality in Hierarchial N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. Journal of Parallel and Distributed Computing, 1995 <<complete citation>>.

[SJG+93] Singh, J.P., Joe T., Gupta, A. and Hennessy, J. An Empirical Comparison of the KSR-1 and DASH Multiprocessors. Proceedings of Supercomputing'93, November 1993.

[SRG94] Jaswinder Pal Singh, Edward Rothberg and Anoop Gupta. Modeling Communication in Parallel Algorithms: A Fruitful Interaction Between Theory and Systems? In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[SWG91] Singh, J.P., Weber, W-D., and Gupta, A. SPLASH: The Stanford ParalleL Applications for Shared Memory. Computer Architecture News, vol. 20, no. 1, pp. 5-44, March 1992.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM, vol. 33, no. 8, August 1990, pp. 103-111.

3.10 Exercises

3.1 Short Answer Questions:

a. For which of the applications that we have described (Ocean, Galaxy, Raytrace) can we be described to have followed this view of decomposing data rather than computation and using an owner-computes rule in our parallelization? What would be the problem(s) with using a strict data distribution and owner-computes rule in the others? How would you address the problem(s)?

b. What are the advantages and disadvantages of using distributed task queues (as opposed to a global task queue) to implement load balancing?

c. Do small tasks inherently increase communication, contention and task management overhead?

d. Draw *one* arc from each kind of memory system traffic below (the list on the left) to the solution technique (on the right) that is *the most effective way* to reduce that source of traffic in a machine that supports a shared address space with physically distributed memory.

Kinds of Memory System Traffic	Solution Techniques
Cold-Start Traffic	Large Cache Sizes
Inherent Communication	Data Placement
Extra Data Communication on a Miss	Algorithm Reorganization
Capacity-Generated Communication	Larger Cache Block Size
Capacity-Generated Local Traffic	Data Structure Reorganization

e. Under what conditions would the sum of busy-useful time across processes (in the execution time) not equal the busy-useful time for the sequential program, assuming both the sequential and parallel programs are deterministic? Provide examples.

3.2 Levels of parallelism.

a. As an example of hierarchical parallelism, consider an algorithm frequently used in medical diagnosis and economic forecasting. The algorithm propagates information through a network or graph, such as the one in Figure 3-22. Every node represents a matrix of values. The arcs correspond to dependences between nodes, and are the channels along which information must flow. The algorithm starts from the nodes at the bottom of the graph and works upward, performing matrix operations at every node encountered along the way. It affords parallelism at least two levels: Nodes that do not have an ancestor-descendent relationship in the traversal can be computed in parallel, and the matrix computations within a node can be parallelized as well. How would you parallelize this algorithm? What are the tradeoffs that are most important?

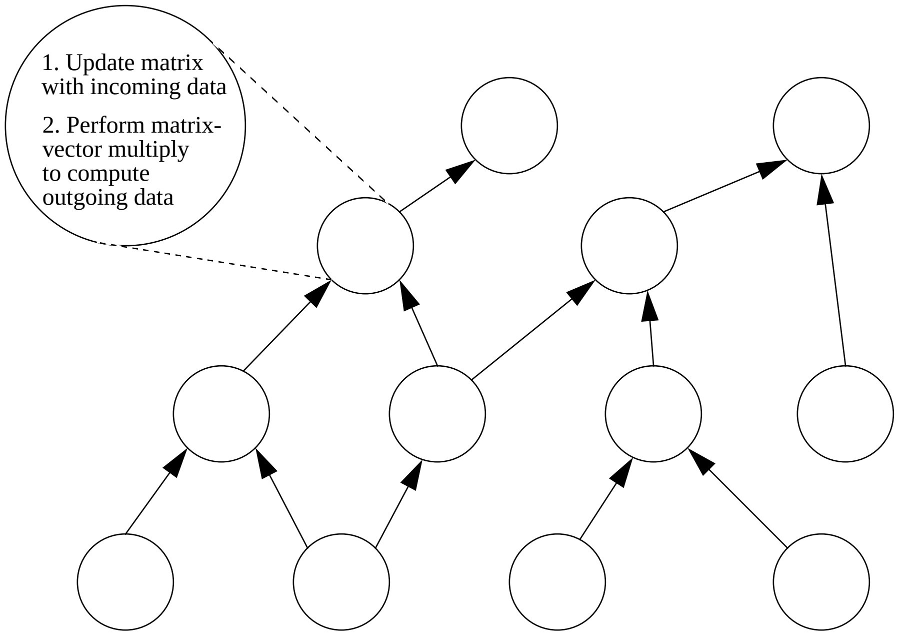

1. Update matrix with incoming data
2. Perform matrix-vector multiply to compute outgoing data

Figure 3-22 Levels of parallelism in a graph computation.

The work within a graph node is shown in the expanded node o the left.

b. Levels of parallelism in locusroute. What are the tradeoffs in determining which level to pick. What parameters affect your decision? What you would you pick for this case: 30 wires, 24 processors, 5 segments per wire, 10 routes per segment, each route evaluation takes same amount of time? If you had to pick one and be tied to it for all cases, which would you pick (you can make and state reasonable assumptions to guide your answer)?

3.3 LU factorization.

a. Analyze the load balance and communication volume for both cases of broadcast based LU with row decomposition: block assignment and interleaved assignment.

b. What if we used a two-dimensional scatter decomposition at the granularity of individual elements. Analyze the load balance and communication volume.

c. Discuss the tradeoffs (programming difficulty and likely performance differences) in programming the different versions in the different communication abstractions. Would you expect one to always be better?

d. Can you think of a better decomposition and assignment? Discuss.

e. Analyze the load balance and communication volume for a pipelined implementation in a 2-d scatter decomposition of elements.

f. Discuss the tradeoffs in the performance of the broadcast versus pipelined versions.

3.4 If E is the set of sections of the algorithm that are enhanced through parallelism, f_k is the fraction of the sequential execution time taken up by the k^{th} enhanced section when run on a uniprocessor, and s_k is the speedup obtained through parallelism on the k^{th} enhanced section, derive an expression for the overall speedup obtained. Apply it to the broadcast approach for

LU factorization at element granularity. Draw a rough concurrency profile for the computation (a graph showing the amount of concurrency versus time, where the unit of time is a logical operation, say updating an interior active element, performed on one or more processors. Assume a 100-by-100 element matrix. Estimate the speedup, ignoring memory referencing and communication costs.

3.5 We have discussed the technique of blocking that is widely used in linear algebra algorithms to exploit temporal locality (see Section 3.4.1). Consider a sequential LU factorization program as described in this chapter. Compute the read miss rate on a system with a cache size of 16KB and a matrix size of 1024-by-1024. Ignore cache conflicts, and count only access to matrix elements. Now imagine that LU factorization is blocked for temporal locality, and compute the miss rate for a sequential implementation (this may be best done after reading the section describing the LU factorization program in the next chapter). Assume a block size of 32-by-32 elements, and that the update to a B-by-B block takes B^3 operations and B^2 cache misses. Assume no reuse of blocks across block operations. If read misses cost 50 cycles, what is the performance difference between the two versions (counting operations as defined above and ignoring write accesses).

3.6 It was mentioned in the chapter that termination detection is an interesting aspect of task stealing. Consider a task stealing scenario in which processes produce tasks as the computation is ongoing. Design a good tasking method (where to take tasks from, how to put tasks into the pool, etc.) and think of some good termination detection heuristics. Perform worst-case complexity analysis for the number of messages needed by the termination detection methods you consider. Which one would you use in practice? Write pseudocode for one that is guaranteed to work and should yield good performance

3.7 Consider transposing a matrix in parallel, from a source matrix to a destination matrix, i.e. B[i,j] = A[j,i].

a. How might you partition the two matrices among processes? Discuss some possibilities and the tradeoffs. Does it matter whether you are programming a shared address space or message passing machine?

b. Why is the interprocess communication in a matrix transpose called all-to-all personalized communication?

c. Write simple pseudocode for the parallel matrix transposition in a shared address space and in message passing (just the loops that implement the transpose). What are the major performance issues you consider in each case, other than inherent communication and load balance, and how do you address them?

d. Is there any benefit to blocking the parallel matrix transpose? Under what conditions, and how would you block it (no need to write out the full code)? What, if anything, is the difference between blocking here and in LU factorization?

3.8 The communication needs of applications, even expressed in terms of bytes per instruction, can help us do back-of-the-envelope calculations to determine the impact of increased bandwidth or reduced latency. For example, a Fast Fourier Transform (FFT) is an algorithm that is widely used in digital signal processing and climate modeling applications. A simple parallel FFT on n data points has a per-process computation cost of $O(n \log n / p)$, and per-process communication volume of $O(n/p)$, where p is the number of processes. The communication to computation ratio is therefore $O(1/\log n)$. Suppose for simplicity that all the constants in the above expressions are unity, and that we are performing an n=1M (or 2^{20}) point FFT on p=1024 processes. Let the average communication latency for a word of data (a point in the FFT) be 200 processor cycles, and let the communication bandwidth between any node and the network be 100MB/sec. Assume no load imbalance or synchronization costs.

a. With no latency hidden, for what fraction of the execution time is a process stalled due to communication latency?

b. What would be the impact on execution time of halving the communication latency?

c. What are the node-to-network bandwidth requirements without latency hiding?

d. What are the node-to-network bandwidth requirements assuming all latency is hidden, and does the machine satisfy them?

3.9 What kind of data (local, nonlocal, or both) constitute the relevant working set in: (i) main memory in a message-passing abstraction, (ii) processor cache in a message-passing abstraction, (iii) processor cache in cache-coherent shared address space abstraction.

3.10 Implement a reduction and a broadcast. First an O(p) linear method, then an O(log p) i.e. tree based, method. Do this both for a shared address space and for message-passing.

3.11 After the assignment of tasks to processors, there is still the issue of scheduling the tasks that a process is assigned to run in some temporal order. What are the major issues involved here? Which are the same as on uniprocessor programs, and which are different? Construct examples that highlight the impact of poor scheduling in the different cases.

3.12 In the data mining case study, why are 2-itemsets computed from the original format of the database rather than the transformed format?

3.13 You have been given the job of creating a word count program for a major book publisher. You will be working on a shared-memory multiprocessor with 32 processing elements. Your only stated interface is “get_words” which returns an array of the book’s next 1000 words to be counted. The main work each processor will do should look like this:

```
while(get_words(word)) {
   for (i=0; i<1000; i++) {
      if word[i] is in list
         increment its count
      else
         add word to list
   }
}
/* Once all words have been logged,
the list should printed out */
```

Using pseudocode, create a detailed description of the control flow and data structures that you will use for this program. Your method should attempt to minimize space, synchronization overhead, and memory latency. This problem allows you a lot of flexibility, so state all assumptions and design decisions.