CHAPTER 4

The *Algorithm Structure* Design Space

4.1 INTRODUCTION

The first phase of designing a parallel algorithm consists of analyzing the problem to identify exploitable concurrency, usually by using the patterns of the *Finding Concurrency* design space. The output from the *Finding Concurrency* design space is a decomposition of the problem into design elements:

- A task decomposition that identifies tasks that can execute concurrently
- A data decomposition that identifies data local to each task
- A way of grouping tasks and ordering the groups to satisfy temporal constraints
- An analysis of dependencies among tasks

These elements provide the connection from the *Finding Concurrency* design space to the *Algorithm Structure* design space. Our goal in the *Algorithm Structure* design space is to refine the design and move it closer to a program that can execute tasks concurrently by mapping the concurrency onto multiple UEs running on a parallel computer.

Of the countless ways to define an algorithm structure, most follow one of six basic design patterns. These patterns make up the *Algorithm Structure* design space. An overview of this design space and its place in the pattern language is shown in Fig. 4.1.

The key issue at this stage is to decide which pattern or patterns are most appropriate for the problem.

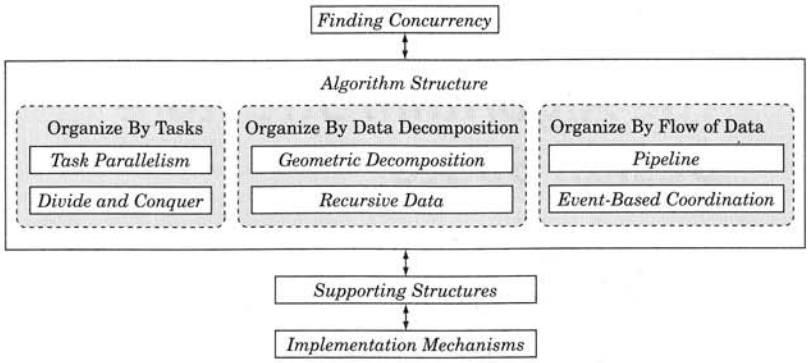

Figure 4.1: Overview of the *Algorithm Structure* design space and its place in the pattern language

First of all, we need to keep in mind that different aspects of the analysis can pull the design in different directions; one aspect might suggest one structure while another suggests a different structure. In nearly every case, however, the following forces should be kept in mind.

- **Efficiency.** It is crucial that a parallel program run quickly and make good use of the computer resources.
- **Simplicity.** A simple algorithm resulting in easy-to-understand code is easier to develop, debug, verify, and modify.
- **Portability.** Ideally, programs should run on the widest range of parallel computers. This will maximize the "market" for a particular program. More importantly, a program is used for many years, while any particular computer system is used for only a few years. Portable programs protect a software investment.
- **Scalability.** Ideally, an algorithm should be effective on a wide range of numbers of processing elements (PEs), from a few up to hundreds or even thousands.

These forces conflict in several ways, however.

Efficiency conflicts with portability: Making a program efficient almost always requires that the code take into account the characteristics of the specific system on which it is intended to run, which limits portability. A design that makes use of the special features of a particular system or programming environment may lead to an efficient program for that particular environment, but be unusable for a different platform, either because it performs poorly or because it is difficult or even impossible to implement for the new platform.

Efficiency also can conflict with simplicity: For example, to write efficient programs that use the *Task Parallelism* pattern, it is sometimes necessary to use

complicated scheduling algorithms. These algorithms in many cases, however, make the program very difficult to understand.

Thus, a good algorithm design must strike a balance between (1) abstraction and portability and (2) suitability for a particular target architecture. The challenge faced by the designer, especially at this early phase of the algorithm design, is to leave the parallel algorithm design abstract enough to support portability while ensuring that it can eventually be implemented effectively for the parallel systems on which it will be executed.

4.2 CHOOSING AN *ALGORITHM STRUCTURE* PATTERN

Finding an effective *Algorithm Structure* pattern for a given problem can be accomplished by considering the questions in the following sections.

4.2.1 Target Platform

What constraints are placed on the parallel algorithm by the target machine or programming environment?

In an ideal world, it would not be necessary to consider the details of the target platform at this stage of the design, because doing so works against keeping the program portable and scalable. This is not an ideal world, however, and software designed without considering the major features of the target platform is unlikely to run efficiently.

The primary issue is how many units of execution (UEs) the system will effectively support, because an algorithm that works well for ten UEs may not work well for hundreds of UEs. It is not necessary to decide on a specific number (in fact to do so would overly constrain the applicability of the design), but it is important to have in mind at this point an order of magnitude for the number of UEs.

Another issue is how expensive it is to share information among UEs. If there is hardware support for shared memory, information exchange takes place through shared access to common memory, and frequent data sharing makes sense. If the target is a collection of nodes connected by a slow network, however, the communication required to share information is very expensive and must be avoided wherever possible.

When thinking about both of these issues—the number of UEs and the cost of sharing information—avoid the tendency to over-constrain the design. Software typically outlives hardware, so over the course of a program's life it may be used on a tremendous range of target platforms. The goal is to obtain a design that works well on the original target platform, but at the same time is flexible enough to adapt to different classes of hardware.

Finally, in addition to multiple UEs and some way to share information among them, a parallel computer has one or more programming environments that can be used to implement parallel algorithms. Different programming environments provide different ways to create tasks and share information among UEs, and a design that does not map well onto the characteristics of the target programming environment will be difficult to implement.

4.2.2 Major Organizing Principle

When considering the concurrency in the problem, is there a particular way of looking at it that stands out and provides a high-level mechanism for organizing this concurrency?

The analysis carried out using the patterns of the *Finding Concurrency* design space describes the potential concurrency in terms of tasks and groups of tasks, data (both shared and task-local), and ordering constraints among task groups. The next step is to find an algorithm structure that represents how this concurrency maps onto the UEs. There is usually a *major organizing principle* implied by the concurrency. This usually falls into one of three camps: *organization by tasks, organization by data decomposition,* and *organization by flow of data.* We now consider each of these in more detail.

For some problems, there is really only one group of tasks active at one time, and the way the tasks within this group interact is the major feature of the concurrency. Examples include so-called *embarrassingly parallel* programs in which the tasks are completely independent, as well as programs in which the tasks in a single group cooperate to compute a result.

For other problems, the way data is decomposed and shared among tasks stands out as the major way to organize the concurrency. For example, many problems focus on the update of a few large data structures, and the most productive way to think about the concurrency is in terms of how this structure is decomposed and distributed among UEs. Programs to solve differential equations or carry out linear algebra computations often fall into this category because they are frequently based on updating large data structures.

Finally, for some problems, the major feature of the concurrency is the presence of well-defined interacting groups of tasks, and the key issue is how the data flows among the tasks. For example, in a signal-processing application, data may flow through a sequence of tasks organized as a pipeline, each performing a transformation on successive data elements. Or a discrete-event simulation might be parallelized by decomposing it into a tasks interacting via "events". Here, the major feature of the concurrency is the way in which these distinct task groups interact.

Notice also that the most effective parallel algorithm design might make use of multiple algorithm structures (combined hierarchically, compositionally, or in sequence), and this is the point at which to consider whether such a design makes sense. For example, it often happens that the very top level of the design is a sequential composition of one or more *Algorithm Structure* patterns. Other designs might be organized hierarchically, with one pattern used to organize the interaction of the major task groups and other patterns used to organize tasks within the groups—for example, an instance of the *Pipeline* pattern in which individual stages are instances of the *Task Parallelism* pattern.

4.2.3 The *Algorithm Structure* Decision Tree

For each subset of tasks, which *Algorithm Structure* design pattern most effectively defines how to map the tasks onto UEs?

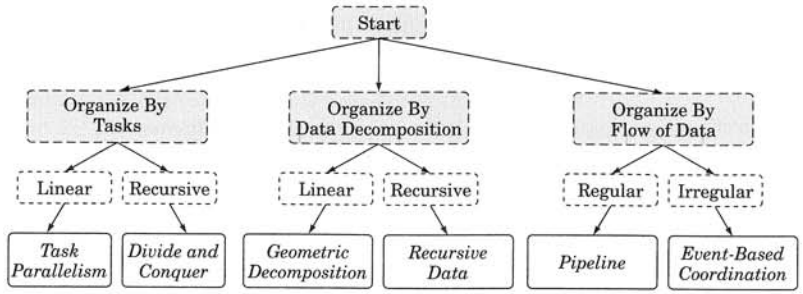

Figure 4.2: Decision tree for the *Algorithm Structure* design space

Having considered the questions raised in the preceding sections, we are now ready to select an algorithm structure, guided by an understanding of constraints imposed by the target platform, an appreciation of the role of hierarchy and composition, and a major organizing principle for the problem. The decision is guided by the decision tree shown in Fig. 4.2. Starting at the top of the tree, consider the concurrency and the major organizing principle, and use this information to select one of the three branches of the tree; then follow the upcoming discussion for the appropriate subtree. Notice again that for some problems, the final design might combine more than one algorithm structure: If no single structure seems suitable, it might be necessary to divide the tasks making up the problem into two or more groups, work through this procedure separately for each group, and then determine how to combine the resulting algorithm structures.

Organize By Tasks. Select the *Organize By Tasks* branch when the execution of the tasks themselves is the best organizing principle. Then determine how the tasks are enumerated. If they can be gathered into a set linear in any number of dimensions, choose the *Task Parallelism* pattern. This pattern includes both situations in which the tasks are independent of each other (so-called embarrassingly parallel algorithms) and situations in which there are some dependencies among the tasks in the form of access to shared data or a need to exchange messages. If the tasks are enumerated by a recursive procedure, choose the *Divide and Conquer* pattern. In this pattern, the problem is solved by recursively dividing it into subproblems, solving each subproblem independently, and then recombining the subsolutions into a solution to the original problem.

Organize By Data Decomposition. Select the *Organize By Data Decomposition* branch when the decomposition of the data is the major organizing principle in understanding the concurrency. There are two patterns in this group, differing in how the decomposition is structured—linearly in each dimension or recursively. Choose the *Geometric Decomposition* pattern when the problem space is decomposed into discrete subspaces and the problem is solved by computing solutions for the subspaces, with the solution for each subspace typically requiring data from a small number of other subspaces. Many instances of this pattern can be found in

scientific computing, where it is useful in parallelizing grid-based computations, for example. Choose the *Recursive Data* pattern when the problem is defined in terms of following links through a recursive data structure (for example, a binary tree).

Organize By Flow of Data. Select the *Organize By Flow of Data* branch when the major organizing principle is how the flow of data imposes an ordering on the groups of tasks. This pattern group has two members, one that applies when this ordering is regular and static and one that applies when it is irregular and/or dynamic. Choose the *Pipeline* pattern when the flow of data among task groups is regular, one-way, and does not change during the algorithm (that is, the task groups can be arranged into a pipeline through which the data flows). Choose the *Event-Based Coordination* pattern when the flow of data is irregular, dynamic, and/or unpredictable (that is, when the task groups can be thought of as interacting via asynchronous events).

4.2.4 Re-evaluation

Is the *Algorithm Structure* pattern (or patterns) suitable for the target platform? It is important to frequently review decisions made so far to be sure the chosen pattern(s) are a good fit with the target platform.

After choosing one or more *Algorithm Structure* patterns to be used in the design, skim through their descriptions to be sure they are reasonably suitable for the target platform. (For example, if the target platform consists of a large number of workstations connected by a slow network, and one of the chosen *Algorithm Structure* patterns requires frequent communication among tasks, it might be difficult to implement the design efficiently.) If the chosen patterns seem wildly unsuitable for the target platform, try identifying a secondary organizing principle and working through the preceding step again.

4.3 EXAMPLES

4.3.1 Medical Imaging

For example, consider the medical imaging problem described in Sec. 3.1.3. This application simulates a large number of gamma rays as they move through a body and out to a camera. One way to describe the concurrency is to define the simulation of each ray as a task. Because they are all logically equivalent, we put them into a single task group. The only data shared among the tasks is a large data structure representing the body, and since access to this data structure is read-only, the tasks do not depend on each other.

Because there are many independent tasks for this problem, it is less necessary than usual to consider the target platform: The large number of tasks should mean that we can make effective use of any (reasonable) number of UEs; the independence of the tasks should mean that the cost of sharing information among UEs will not have much effect on performance.

Thus, we should be able to choose a suitable structure by working through the decision tree shown previously in Fig. 4.2. Given that in this problem the tasks are independent, the only issue we really need to worry about as we select an algorithm structure is how to map these tasks onto UEs. That is, for this problem, the major organizing principle seems to be the way the tasks are organized, so we start by following the *Organize By Tasks* branch.

We now consider the nature of our set of tasks—whether they are arranged hierarchically or reside in an unstructured or flat set. For this problem, the tasks are in an unstructured set with no obvious hierarchical structure among them, so we choose the *Task Parallelism* pattern. Note that in the problem, the tasks are independent, a fact that we will be able to use to simplify the solution.

Finally, we review this decision in light of possible target-platform considerations. As we observed earlier, the key features of this problem (the large number of tasks and their independence) make it unlikely that we will need to reconsider because the chosen structure will be difficult to implement on the target platform.

4.3.2 Molecular Dynamics

As a second example, consider the molecular dynamics problem described in Sec. 3.1.3. In the *Task Decomposition* pattern, we identified the following groups of tasks associated with this problem:

- Tasks that find the vibrational forces on an atom
- Tasks that find the rotational forces on an atom
- Tasks that find the nonbonded forces on an atom
- Tasks that update the position and velocity of an atom
- A task to update the neighbor list for all the atoms

The tasks within each group are expressed as the iterations of a loop over the atoms within the molecular system.

We can choose a suitable algorithm structure by working through the decision tree shown earlier in Fig. 4.2. One option is to organize the parallel algorithm in terms of the flow of data among the groups of tasks. Note that only the first three task groups (the vibrational, rotational, and nonbonded force calculations) can execute concurrently; that is, they must finish computing the forces before the atomic positions, velocities and neighbor lists can be updated. This is not very much concurrency to work with, so a different branch in Fig. 4.2 should be used for this problem.

Another option is to derive exploitable concurrency from the set of tasks within each group, in this case the iterations of a loop over atoms. This suggests an organization by tasks with a linear arrangement of tasks, or based on Fig. 4.2, the *Task Parallelism* pattern should be used. Total available concurrency is large (on the order of the number of atoms), providing a great deal of flexibility in designing the parallel algorithm.

The target machine can have a major impact on the parallel algorithm for this problem. The dependencies discussed in the *Data Decomposition* pattern (replicated coordinates on each UE and a combination of partial sums from each UE to compute

a global force array) suggest that on the order of $2 \cdot 3 \cdot N$ terms (where N is the number of atoms) will need to be passed among the UEs. The computation, however, is of order $n \cdot N$, where n is the number of atoms in the neighborhood of each atom and considerably less than N. Hence, the communication and computation are of the same order and management of communication overhead will be a key factor in designing the algorithm.

4.4 THE *TASK PARALLELISM* PATTERN

Problem

When the problem is best decomposed into a collection of tasks that can execute concurrently, how can this concurrency be exploited efficiently?

Context

Every parallel algorithm is fundamentally a collection of concurrent tasks. These tasks and any dependencies among them can be identified by inspection (for simple problems) or by application of the patterns in the *Finding Concurrency* design space. For some problems, focusing on these tasks and their interaction might not be the best way to organize the algorithm: In some cases it makes sense to organize the tasks in terms of the data (as in the *Geometric Decomposition* pattern) or the flow of data among concurrent tasks (as in the *Pipeline* pattern). However, in many cases it is best to work directly with the tasks themselves. When the design is based directly on the tasks, the algorithm is said to be a *task parallel* algorithm.

The class of task parallel algorithms is very large. Examples include the following.

- Ray-tracing codes such as the medical-imaging example described in the *Task Decomposition* pattern: Here the computation associated with each "ray" becomes a separate and completely independent task.
- The molecular-dynamics example described in the *Task Decomposition* pattern: The update of the nonbonded force on each atom is a task. The dependencies among tasks are managed by replicating the force array on each UE to hold the partial sums for each atom. When all the tasks have completed their contributions to the nonbonded force, the individual force arrays are combined (or "reduced") into a single array holding the full summation of nonbonded forces for each atom.
- Branch-and-bound computations, in which the problem is solved by repeatedly removing a solution space from a list of such spaces, examining it, and either declaring it a solution, discarding it, or dividing it into smaller solution spaces that are then added to the list of spaces to examine. Such computations can be parallelized using this pattern by making each "examine and process a solution space" step a separate task. The tasks weakly depend on each other through the shared queue of tasks.

The common factor is that the problem can be decomposed into a collection of tasks that can execute concurrently. The tasks can be completely independent (as in the medical-imaging example) or there can be dependencies among them (as in the molecular-dynamics example). In most cases, the tasks will be associated with iterations of a loop, but it is possible to associate them with larger-scale program structures as well.

In many cases, all of the tasks are known at the beginning of the computation (the first two examples). However, in some cases, tasks arise dynamically as the computation unfolds, as in the branch-and-bound example.

Also, while it is usually the case that all tasks must be completed before the problem is done, for some problems, it may be possible to reach a solution without completing all of the tasks. For example, in the branch-and-bound example, we have a pool of tasks corresponding to solution spaces to be searched, and we might find an acceptable solution before all the tasks in this pool have been completed.

Forces

- To exploit the potential concurrency in the problem, we must assign tasks to UEs. Ideally we want to do this in a way that is simple, portable, scalable, and efficient. As noted in Sec. 4.1, however, these goals may conflict. A key consideration is balancing the load, that is, ensuring that all UEs have roughly the same amount of work to do.
- If the tasks depend on each other in some way (via either ordering constraints or data dependencies), these dependencies must be managed correctly, again keeping in mind the sometimes-conflicting goals of simplicity, portability, scalability, and efficiency.

Solution

Designs for task-parallel algorithms involve three key elements: the tasks and how they are defined, the dependencies among them, and the *schedule* (how the tasks are assigned to UEs). We discuss them separately, but in fact they are tightly coupled, and all three must be considered before final decisions are made. After these factors are considered, we look at the overall program structure and then at some important special cases of this pattern.

Tasks. Ideally, the tasks into which the problem is decomposed should meet two criteria: First, there should be at least as many tasks as UEs, and preferably many more, to allow greater flexibility in scheduling. Second, the computation associated with each task must be large enough to offset the overhead associated with managing the tasks and handling any dependencies. If the initial decomposition does not meet these criteria, it is worthwhile to consider whether there is another way of decomposing the problem into tasks that does meet the criteria.

For example, in image-processing applications where each pixel update is independent, the task definition can be individual pixels, image lines, or even whole blocks in the image. On a system with a small number of nodes connected by

a slow network, tasks should be large to offset high communication latencies, so basing tasks on blocks of the image is appropriate. The same problem on a system containing a large number of nodes connected by a fast (low-latency) network, however, would need smaller tasks to make sure enough work exists to keep all the UEs occupied. Notice that this imposes a requirement for a fast network, because otherwise the smaller amount of work per task will not be enough to compensate for communication overhead.

Dependencies. Dependencies among tasks have a major impact on the emerging algorithm design. There are two categories of dependencies, ordering constraints and dependencies related to shared data.

For this pattern, ordering constraints apply to task groups and can be handled by forcing the groups to execute in the required order. For example, in a task-parallel multidimensional Fast Fourier Transform, there is a group of tasks for each dimension of the transform, and synchronization or other program constructs are used to make sure computation on one dimension completes before the next dimension begins. Alternatively, we could simply think of such a problem as a sequential composition of task-parallel computations, one for each task group.

Shared-data dependencies are potentially more complicated. In the simplest case, there are no dependencies among the tasks. A surprisingly large number of problems can be cast into this form. Such problems are often called embarrassingly parallel. Their solutions are among the simplest of parallel programs; the main considerations are how the tasks are defined (as discussed previously) and scheduled (as discussed later). When data is shared among tasks, the algorithm can be much more complicated, although there are still some common cases that can be dealt with relatively easily. We can categorize dependencies as follows.

- **Removable dependencies.** In this case, the dependency is not a true dependency between tasks, but an apparent dependency that can be removed by simple code transformations. The simplest case is a temporary variable whose use is completely local to each task; that is, each task initializes the variable without reference to other tasks. This case can be handled by simply creating a copy of the variable local to each UE. In more complicated cases, iterative expressions might need to be transformed into closed-form expressions to remove a loop-carried dependency. For example, consider the following simple loop:

```
int ii = 0, jj = 0;

for(int i = 0; i< N; i++)
{
   ii = ii + 1;
   d[ii] = big_time_consuming_work(ii);
   jj = jj + i;
   a[jj] = other_big_calc(jj);
}
```


The variables `ii` and `jj` create a dependency between tasks and prevent parallelization of the loop. We can remove this dependency by replacing `ii` and `jj` with closed-form expressions (noticing that the values of `ii` and `i` are the same and that the value of `jj` is the sum of the values from 0 through `i`):

```
for(int i = 0; i< N; i++){

   d[i] = big_time_consuming_work(i);
   a[(i*i+i)/2] = other_big_calc((i*i+i)/2));
}
```

- **"Separable" dependencies.** When the dependencies involve accumulation into a shared data structure, they can be separated from the tasks ("pulled outside the concurrent computation") by replicating the data structure at the beginning of the computation, executing the tasks, and then combining the copies into a single data structure after the tasks complete. Often the accumulation is a *reduction* operation, in which a collection of data elements is reduced to a single element by repeatedly applying a binary operation such as addition or multiplication.

  In more detail, these dependencies can be managed as follows: A copy of the data structure used in the accumulation is created on each UE. Each copy is initialized (in the case of a reduction, to the identity element for the binary operation—for example, zero for addition and one for multiplication). Each task then carries out the accumulation into its local data structure, eliminating the shared-data dependency. When all tasks are complete, the local data structures on each UE are combined to produce the final global result (in the case of a reduction, by applying the binary operation again). As an example, consider the following loop to sum the elements of array `f`:

```
for(int i = 0; i< N; i++){
   sum = sum + f(i);
}
```

  This is technically a dependency between loop iterations, but if we recognize that the loop body is just accumulating into a simple scalar variable, it can be handled as a reduction.

  Reductions are so common that both MPI and OpenMP provide support for them as part of the API. Sec. 6.4.2 in the *Implementation Mechanisms* design space discusses reductions in more detail.

- **Other dependencies.** If the shared data cannot be pulled out of the tasks and is both read and written by the tasks, data dependencies must be

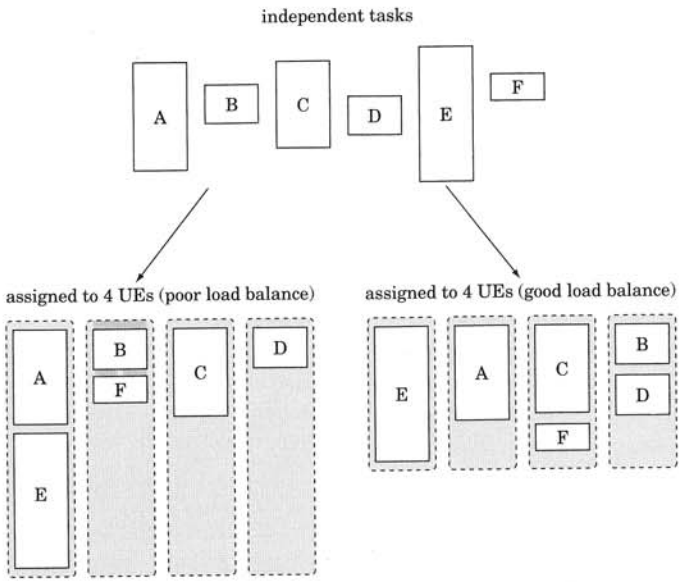

Figure 4.3: Good versus poor load balance

explicitly managed within the tasks. How to do this in a way that gives correct results and also acceptable performance is the subject of the *Shared Data* pattern.

Schedule. The remaining key element to consider is the *schedule*—the way in which tasks are assigned to UEs and scheduled for execution. Load balance (as described in Chapter 2) is a critical consideration in scheduling; a design that balances the computational load among PEs will execute more efficiently than one that does not. Fig. 4.3 illustrates the problem.

Two classes of schedules are used in parallel algorithms: static schedules, in which the distribution of tasks among UEs is determined at the start of the computation and does not change; and dynamic schedules, in which the distribution of tasks among UEs varies as the computation proceeds.

In a static schedule, the tasks are associated into blocks and then assigned to UEs. Block size is adjusted so each UE takes approximately the same amount of time to complete its tasks. In most applications using a static schedule, the computational resources available from the UEs are predictable and stable over the course of the computation, with the most common case being UEs that are identical (that is, the computing system is homogeneous). If the set of times required to complete each task is narrowly distributed about a mean, the sizes of the blocks should be proportional to the relative performance of the UEs (so, in a homogeneous

system, they are all the same size). When the effort associated with the tasks varies considerably, a static schedule can still be useful, but now the number of blocks assigned to UEs must be much greater than the number of UEs. By dealing out the blocks in a round-robin manner (much as a deck of cards is dealt among a group of card players), the load is balanced statistically.

Dynamic schedules are used when (1) the effort associated with each task varies widely and is unpredictable and/or (2) when the capabilities of the UEs vary widely and unpredictably. The most common approach used for dynamic load balancing is to define a task queue to be used by all the UEs; when a UE completes its current task and is therefore ready to process more work, it removes a task from the task queue. Faster UEs or those receiving lighter-weight tasks will access the queue more often and thereby be assigned more tasks.

Another dynamic scheduling strategy uses *work stealing*, which works as follows. The tasks are distributed among the UEs at the start of the computation. Each UE has its own work queue. When the queue is empty, the UE will try to steal work from the queue on some other UE (where the other UE is usually randomly selected). In many cases, this produces an optimal dynamic schedule without incurring the overhead of maintaining a single global queue. In programming environments or packages that provide support for the construct, such as Cilk [BJK+96], Hood [BP99], or the FJTask framework [Lea00b, Lea], it is straightforward to use this approach. But with more commonly used programming environments such as OpenMP, MPI, or Java (without support such as the FJTask framework), this approach adds significant complexity and therefore is not often used.

Selecting a schedule for a given problem is not always easy. Static schedules incur the least overhead during the parallel computation and should be used whenever possible.

Before ending the discussion of schedules, we should mention again that while for most problems all of the tasks are known when the computation begins and all must be completed to produce an overall solution, there are problems for which one or both of these is not true. In these cases, a dynamic schedule is probably more appropriate.

Program structure. Many task-parallel problems can be considered to be loop-based. Loop-based problems are, as the name implies, those in which the tasks are based on the iterations of a loop. The best solutions for such problems use the *Loop Parallelism* pattern. This pattern can be particularly simple to implement in programming environments that provide directives for automatically assigning loop iterations to UEs. For example, in OpenMP a loop can be parallelized by simply adding a "parallel for" directive with an appropriate schedule clause (one that maximizes efficiency). This solution is especially attractive because OpenMP then guarantees that the resulting program is semantically equivalent to the analogous sequential code (within roundoff error associated with different orderings of floating-point operations).

For problems in which the target platform is not a good fit with the *Loop Parallelism* pattern, or for problems in which the model of "all tasks known initially, all tasks must complete" does not apply (either because tasks can be created during

the computation or because the computation can terminate without all tasks being complete), this straightforward approach is not the best choice. Instead, the best design makes use of a *task queue;* tasks are placed on the task queue as they are created and removed by UEs until the computation is complete. The overall program structure can be based on either the *Master/Worker* pattern or the *SPMD* pattern. The former is particularly appropriate for problems requiring a dynamic schedule.

In the case in which the computation can terminate before all the tasks are complete, some care must be taken to ensure that the computation ends when it should. If we define the *termination condition* as the condition that when true means the computation is complete—either all tasks are complete or some other condition (for example, an acceptable solution has been found by one task)—then we want to be sure that (1) the termination condition is eventually met (which, if tasks can be created dynamically, might mean building into it a limit on the total number of tasks created), and (2) when the termination condition is met, the program ends. How to ensure the latter is discussed in the *Master/Worker* and *SPMD* patterns.

Common idioms. Most problems for which this pattern is applicable fall into the following two categories.

Embarrassingly parallel problems are those in which there are no dependencies among the tasks. A wide range of problems fall into this category, ranging from rendering frames in a motion picture to statistical sampling in computational physics. Because there are no dependencies to manage, the focus is on scheduling the tasks to maximize efficiency. In many cases, it is possible to define schedules that automatically and dynamically balance the load among UEs.

Replicated data or reduction problems are those in which dependencies can be managed by "separating them from the tasks" as described earlier—replicating the data at the beginning of computation and combining results when the termination condition is met (usually "all tasks complete"). For these problems, the overall solution consists of three phases, one to replicate the data into local variables, one to solve the now-independent tasks (using the same techniques used for embarrassingly parallel problems), and one to recombine the results into a single result.

Examples

We will consider two examples of this pattern. The first example, an image-construction example, is embarrassingly parallel. The second example will build on the molecular dynamics example used in several of the *Finding Concurrency* patterns.

Image construction. In many image-construction problems, each pixel in the image is independent of all the other pixels. For example, consider the well known Mandelbrot set [Dou86]. This famous image is constructed by coloring each pixel according to the behavior of the quadratic recurrence relation

$$Z_{n+1} = Z_n^2 + C \tag{4.1}$$

where C and Z are complex numbers and the recurrence is started with $Z_0 = C$. The image plots the imaginary part of C on the vertical axis and the real part on the horizontal axis. The color of each pixel is black if the recurrence relation converges to a stable value or is colored depending on how rapidly the relation diverges.

At the lowest level, the task is the update for a single pixel. First consider computing this set on a cluster of PCs connected by an Ethernet. This is a coarse-grained system; that is, the rate of communication is slow relative to the rate of computation. To offset the overhead incurred by the slow network, the task size needs to be large; for this problem, that might mean computing a full row of the image. The work involved in computing each row varies depending on the number of divergent pixels in the row. The variation, however, is modest and distributed closely around a mean value. Therefore, a static schedule with many more tasks than UEs will likely give an effective statistical balance of the load among nodes. The remaining step in applying the pattern is choosing an overall structure for the program. On a shared-memory machine using OpenMP, the *Loop Parallelism* pattern described in the *Supporting Structures* design space is a good fit. On a network of workstations running MPI, the *SPMD* pattern (also in the *Supporting Structures* design space) is appropriate.

Before moving on to the next example, we consider one more target system, a cluster in which the nodes are not heterogeneous—that is, some nodes are much faster than others. Assume also that the speed of each node may not be known when the work is scheduled. Because the time needed to compute the image for a row now depends both on the row and on which node computes it, a dynamic schedule is indicated. This in turn suggests that a general dynamic load-balancing scheme is indicated, which then suggests that the overall program structure should be based on the *Master/Worker* pattern.

Molecular dynamics. For our second example, we consider the computation of the nonbonded forces in a molecular dynamics computation. This problem is described in Sec. 3.1.3 and in [Mat95, PH95] and is used throughout the patterns in the *Finding Concurrency* design space. Pseudocode for this computation is shown in Fig. 4.4. The physics in this example is not relevant and is buried in code not shown here (the computation of the neighbors list and the force function). The basic computation structure is a loop over atoms, and then for each atom, a loop over interactions with other atoms. The number of interactions per atom is computed separately when the neighbors list is determined. This routine (not shown here) computes the number of atoms within a radius equal to a preset cutoff distance. The neighbor list is also modified to account for Newton's third law: Because the force of atom `i` on atom `j` is the negative of the force of atom `j` on atom `i`, only half of the potential interactions need actually be computed. Understanding this detail is not important for understanding this example. The key is that this causes each loop over `j` to vary greatly from one atom to another, thereby greatly complicating the load-balancing problem. Indeed, for the purposes of this example, all that must really be understood is that calculating the force is an expensive operation and that the number of interactions per atom varies greatly. Hence, the computational effort for each iteration over `i` is difficult to predict in advance.

```
function non_bonded_forces (N, Atoms, neighbors, Forces)

    Int const N // number of atoms

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: forces (3,N) //force in each dimension
    Array of List :: neighbors(N) //atoms in cutoff volume
    Real :: forceX, forceY, forceZ

    loop [i] over atoms

        loop [j] over neighbors(i)
            forceX = non_bond_force(atoms(1,i), atoms(1,j))
            forceY = non_bond_force(atoms(2,i), atoms(2,j))
            forceZ = non_bond_force(atoms(3,i), atoms(3,j))
            force(1,i) += forceX; force(1,j) -= forceX;
            force(2,i) += forceY; force(2,j) -= forceY;
            force(3,i) += forceZ; force(3,j) -= forceZ;
        end loop [j]

    end loop [i]
end function non_bonded_forces
```

Figure 4.4: Pseudocode for the nonbonded computation in a typical molecular dynamics code

Each component of the force term is an independent computation, meaning that each (`i`, `j`) pair is fundamentally an independent task. The number of atoms tends to be on the order of thousands, and squaring that gives a number of tasks that is more than enough for all but the largest parallel systems. Therefore, we can take the more convenient approach of defining a task as one iteration of the loop over `i`. The tasks, however, are not independent: The `force` array is read and written by each task. Inspection of the code shows that the arrays are only used to accumulate results from the computation, however. Thus, the full array can be replicated on each UE and the local copies combined (reduced) after the tasks complete.

After the replication is defined, the problem is embarrassingly parallel and the same approaches discussed previously apply. We will revisit this example in the *Master/Worker*, *Loop Parallelism*, and *SPMD* patterns. A choice among these patterns is normally made based on the target platforms.

Known uses. There are many application areas in which this pattern is useful, including the following.

Many ray-tracing programs use some form of partitioning with individual tasks corresponding to scan lines in the final image [BKS91].

Applications written with coordination languages such as Linda are another rich source of examples of this pattern [BCM+91]. Linda [CG91] is a simple language consisting of only six operations that read and write an associative (that is, content-addressable) shared memory called a *tuple space*. The tuple space provides a natural way to implement a wide variety of shared-queue and master/worker algorithms.

Parallel computational chemistry applications also make heavy use of this pattern. In the quantum chemistry program GAMESS, the loops over two electron integrals are parallelized with the task queue implied by the `Nextval` construct within TCGMSG. An early version of the distance geometry program DGEOM was parallelized with the master/worker form of this pattern. These examples are discussed in [Mat95].

PTEP (Parallel Telemetry Processor) [NBB01], developed by NASA as the downlink processing system for data from a planetary rover or lander, also makes use of this pattern. The system is implemented in Java but can incorporate components implemented in other languages. For each incoming data packet, the system determines which instrument produced the data, and then performs an appropriate sequential pipeline of processing steps. Because the incoming data packets are independent, the processing of individual packets can be done in parallel.

4.5 THE *DIVIDE AND CONQUER* PATTERN

Problem

Suppose the problem is formulated using the sequential divide-and-conquer strategy. How can the potential concurrency be exploited?

Context

The divide-and-conquer strategy is employed in many sequential algorithms. With this strategy, a problem is solved by splitting it into a number of smaller subproblems, solving them independently, and merging the subsolutions into a solution for the whole problem. The subproblems can be solved directly, or they can in turn be solved using the same divide-and-conquer strategy, leading to an overall recursive program structure.

This strategy has proven valuable for a wide range of computationally intensive problems. For many problems, the mathematical description maps well onto a divide-and-conquer algorithm. For example, the famous fast Fourier transform algorithm [PTV93] is essentially a mapping of the doubly nested loops of the discrete Fourier transform into a divide-and-conquer algorithm. Less well known is the fact that many algorithms from computational linear algebra, such as the Cholesky decomposition [ABE+97, PLA], also map well onto divide-and-conquer algorithms.

The potential concurrency in this strategy is not hard to see: Because the subproblems are solved independently, their solutions can be computed concurrently. Fig. 4.5 illustrates the strategy and the potential concurrency. Notice that each "split" doubles the available concurrency. Although the concurrency in a divide-and-conquer algorithm is obvious, the techniques required to exploit it effectively are not always obvious.

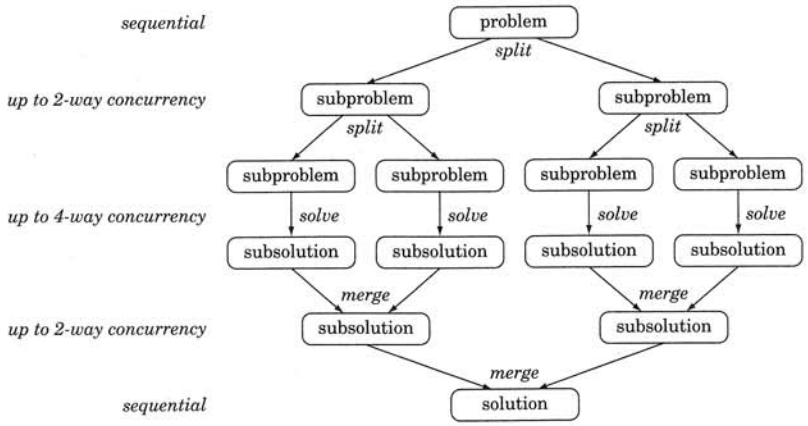

Figure 4.5: The divide-and-conquer strategy

Forces

- The traditional divide-and-conquer strategy is a widely useful approach to algorithm design. Sequential divide-and-conquer algorithms are almost trivial to parallelize based on the obvious exploitable concurrency.
- As Fig. 4.5 suggests, however, the amount of exploitable concurrency varies over the life of the program. At the outermost level of the recursion (initial split and final merge), there is little or no exploitable concurrency, and the subproblems also contain split and merge sections. Amdahl's law (Chapter 2) tells us that the serial parts of a program can significantly constrain the speedup that can be achieved by adding more processors. Thus, if the split and merge computations are nontrivial compared to the amount of computation for the base cases, a program using this pattern might not be able to take advantage of large numbers of processors. Further, if there are many levels of recursion, the number of tasks can grow quite large, perhaps to the point that the overhead of managing the tasks overwhelms any benefit from executing them concurrently.
- In distributed-memory systems, subproblems can be generated on one PE and executed by another, requiring data and results to be moved between the PEs. The algorithm will be more efficient if the amount of data associated with a computation (that is, the size of the parameter set and result for each subproblem) is small. Otherwise, large communication costs can dominate the performance.
- In divide-and-conquer algorithms, the tasks are created dynamically as the computation proceeds, and in some cases, the resulting "task graph" will have an irregular and data-dependent structure. If this is the case, then the solution should employ dynamic load balancing.


```
func solve returns Solution; // a solution stage
func baseCase returns Boolean; // direct solution test
func baseSolve returns Solution; // direct solution
func merge returns Solution; // combine subsolutions
func split returns Problem[]; // split into subprobs

Solution solve(Problem P) {
    if (baseCase(P))
        return baseSolve(P);
    else {
        Problem subProblems[N];
        Solution subSolutions[N];
        subProblems = split(P);
        for (int i = 0; i < N; i++)
            subSolutions[i] = solve(subProblems[i]);
        return merge(subSolutions);
    }
}
```

Figure 4.6: Sequential pseudocode for the divide-and-conquer algorithm

Solution

A sequential divide-and-conquer algorithm has the structure shown in Fig. 4.6. The cornerstone of this structure is a recursively invoked function (`solve()`) that drives each stage in the solution. Inside `solve`, the problem is either split into smaller subproblems (using `split()`) or it is directly solved (using `baseSolve()`). In the classical strategy, recursion continues until the subproblems are simple enough to be solved directly, often with just a few lines of code each. However, efficiency can be improved by adopting the view that `baseSolve()` should be called when (1) the overhead of performing further splits and merges significantly degrades performance, or (2) the size of the problem is optimal for the target system (for example, when the data required for a `baseSolve()` fits entirely in cache).

The concurrency in a divide-and-conquer problem is obvious when, as is usually the case, the subproblems can be solved independently (and hence, concurrently). The sequential divide-and-conquer algorithm maps directly onto a task-parallel algorithm by defining one task for each invocation of the `solve()` function, as illustrated in Fig. 4.7. Note the recursive nature of the design, with each task in effect dynamically generating and then absorbing a task for each subproblem.

At some level of recursion, the amount of computation required for a subproblems can become so small that it is not worth the overhead of creating a new task to solve it. In this case, a hybrid program that creates new tasks at the higher levels of recursion, then switches to a sequential solution when the subproblems become smaller than some threshold, will be more effective. As discussed next, there are tradeoffs involved in choosing the threshold, which will depend on the specifics of the problem and the number of PEs available. Thus, it is a good idea to design the program so that this "granularity knob" is easy to change.

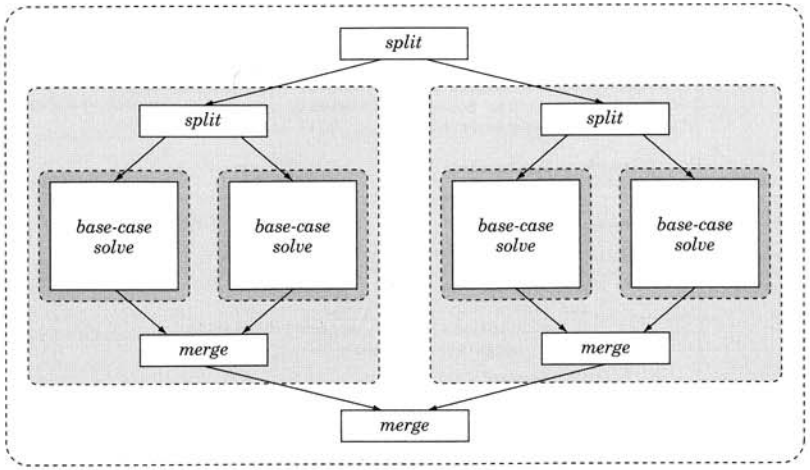

Figure 4.7: Parallelizing the divide-and-conquer strategy. Each dashed-line box represents a task.

Mapping tasks to UEs and PEs. Conceptually, this pattern follows a straightforward fork/join approach (see the *Fork/Join* pattern). One task splits the problem, then forks new tasks to compute the subproblems, waits until the subproblems are computed, and then joins with the subtasks to merge the results.

The easiest situation is when the split phase generates subproblems that are known to be about the same size in terms of needed computation. Then, a straightforward implementation of the fork/join strategy, mapping each task to a UE and stopping the recursion when the number of active subtasks is the same as the number of PEs, works well.

In many situations, the problem will not be regular, and it is best to create more, finer-grained tasks and use a master/worker structure to map tasks to units of execution. This implementation of this approach is described in detail in the *Master/Worker* pattern. The basic idea is to conceptually maintain a queue of tasks and a pool of UEs, typically one per PE. When a subproblem is split, the new tasks are placed in the queue. When a UE finishes a task, it obtains another one from the queue. In this way, all of the UEs tend to remain busy, and the solution shows a good load balance. Finer-grained tasks allow a better load balance at the cost of more overhead for task management.

Many parallel programming environments directly support the fork/join construct. For example, in OpenMP, we could easily produce a parallel application by turning the `for` loop of Fig. 4.6 into an OpenMP `parallel for` construct. Then the subproblems will be solved concurrently rather than in sequence, with the OpenMP runtime environment handling the thread management. Unfortunately, this technique will only work with implementations of OpenMP that support true nesting of parallel regions. Currently, only a few OpenMP implementations do so. Extending OpenMP to better address recursive parallel algorithms is an active area of research

in the OpenMP community [Mat03]. One proposal likely to be adopted in a future OpenMP specification is to add an explicit *taskqueue* construct designed to support the expression of recursive algorithms [SHPT00].

The FJTask framework for Java [Lea00b, Lea] provides support for fork/join programs with a pool of threads backing the implementation. Several example programs using a divide-and-conquer strategy are provided with the package.

Communication costs. Because tasks are generated dynamically from a single top-level task, a task can be executed on a different PE than the one that generated it. In a distributed-memory system, a higher-level task will typically have the data necessary to solve its entire problem, the relevant data must be moved to the subproblem's PE, and the result moved back to the source. Thus it pays to consider how to efficiently represent the parameters and results, and consider whether it makes sense to replicate some data at the beginning of the computation.

Dealing with dependencies. In most algorithms formulated using the divide-and-conquer strategy, the subproblems can be solved independently from each other. Less commonly, the subproblems require access to a common data structure. These dependencies can be handled using the techniques described in the *Shared Data* pattern.

Other optimizations. A factor limiting the scalability of this pattern is the serial split and merge sections. Reducing the number of levels of recursion required by splitting each problem into more subproblems can often help, especially if the split and merge phases can be parallelized themselves. This might require restructuring, but can be quite effective, especially in the limiting case of "one-deep divide and conquer", in which the initial split is into P subproblems, where P is the number of available PEs. Examples of this approach are given in [Tho95].

Examples

Mergesort. Mergesort is a well-known sorting algorithm based on the divide-and-conquer strategy, applied as follows to sort an array of N elements.

- The base case is an array of size less than some threshold. This is sorted using an appropriate sequential sorting algorithm, often quicksort.
- In the split phase, the array is split by simply partitioning it into two contiguous subarrays, each of size $N/2$.
- In the solve-subproblems phase, the two subarrays are sorted (by applying the mergesort procedure recursively).
- In the merge phase, the two (sorted) subarrays are recombined into a single sorted array.

This algorithm is readily parallelized by performing the two recursive mergesorts in parallel.

This example is revisited with more detail in the *Fork/Join* pattern in the *Supporting Structures* design space.

Matrix diagonalization. Dongarra and Sorensen ([DS87]) describe a parallel algorithm for diagonalizing (computing the eigenvectors and eigenvalues of) a symmetric tridiagonal matrix T. The problem is to find a matrix Q such that $Q^T \cdot T \cdot Q$ is diagonal; the divide-and-conquer strategy goes as follows (omitting the mathematical details).

- The base case is a small matrix which is diagonalized sequentially.
- The split phase consists of finding matrix T' and vectors u, v, such that $T = T' + uv^T$, and T' has the form

$$\begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix}$$

  where T_1 and T_2 are symmetric tridiagonal matrices (which can be diagonalized by recursive calls to the same procedure).
- The merge phase recombines the diagonalizations of T_1 and T_2 into a diagonalization of T.

Details can be found in [DS87] or in [GL96].

Known uses. Any introductory algorithms text will have many examples of algorithms based on the divide-and-conquer strategy, most of which can be parallelized with this pattern.

Some algorithms frequently parallelized with this strategy include the Barnes-Hut [BH86] and Fast Multipole [GG90] algorithms used in N-body simulations; signal-processing algorithms, such as discrete Fourier transforms; algorithms for banded and tridiagonal linear systems, such as those found in the ScaLAPACK package [CD97, Sca]; and algorithms from computational geometry, such as convex hull and nearest neighbor.

A particularly rich source of problems that use the *Divide and Conquer* pattern is the FLAME project [GGHvdG01]. This is an ambitious project to recast linear algebra problems in recursive algorithms. The motivation is twofold. First, mathematically, these algorithms are naturally recursive; in fact, most pedagogical discussions of these algorithms are recursive. Second, these recursive algorithms have proven to be particularly effective at producing code that is both portable and highly optimized for the cache architectures of modern microprocessors.

Related Patterns

Just because an algorithm is based on a sequential divide-and-conquer strategy does not mean that it must be parallelized with the *Divide and Conquer* pattern. A hallmark of this pattern is the recursive arrangement of the tasks, leading to a varying amount of concurrency and potentially high overheads on machines for which managing the recursion is expensive. If the recursive decomposition into subproblems can be reused, however, it might be more effective to do the recursive decomposition, and then use some other pattern (such as the *Geometric Decomposition* pattern or the *Task Parallelism* pattern) for the actual computation. For example, the first production-level molecular dynamics program to use the fast multipole method, PMD [Win95], used the *Geometric Decomposition* pattern to parallelize the fast multipole algorithm, even though the original fast multipole algorithm used divide and conquer. This worked because the multipole computation was carried out many times for each configuration of atoms.

4.6 THE *GEOMETRIC DECOMPOSITION* PATTERN

Problem

How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable "chunks"?

Context

Many important problems are best understood as a sequence of operations on a core data structure. There may be other work in the computation, but an effective understanding of the full computation can be obtained by understanding how the core data structures are updated. For these types of problems, often the best way to represent the concurrency is in terms of decompositions of these core data structures. (This form of concurrency is sometimes known as domain decomposition, or coarse-grained data parallelism.)

The way these data structures are built is fundamental to the algorithm. If the data structure is recursive, any analysis of the concurrency must take this recursion into account. For recursive data structures, the *Recursive Data* and *Divide and Conquer* patterns are likely candidates. For arrays and other linear data structures, we can often reduce the problem to potentially concurrent components by decomposing the data structure into contiguous substructures, in a manner analogous to dividing a geometric region into subregions—hence the name *Geometric Decomposition*. For arrays, this decomposition is along one or more dimensions, and the resulting subarrays are usually called blocks. We will use the term *chunks* for the substructures or subregions, to allow for the possibility of more general data structures, such as graphs.

This decomposition of data into chunks then implies a decomposition of the update operation into tasks, where each task represents the update of one chunk, and the tasks execute concurrently. If the computations are strictly local, that is, all required information is within the chunk, the concurrency is embarrassingly parallel and the simpler *Task Parallelism* pattern should be used. In many cases, however, the update requires information from points in other chunks (frequently from what we can call *neighboring chunks*—chunks containing data that was nearby in the original global data structure). In these cases, information must be shared between chunks to complete the update.

Example: mesh-computation program. The problem is to model 1D heat diffusion (that is, diffusion of heat along an infinitely narrow pipe). Initially, the whole pipe is at a stable and fixed temperature. At time 0, we set both ends to different temperatures, which will remain fixed throughout the computation. We then calculate how temperatures change in the rest of the pipe over time. (What we expect is that the temperatures will converge to a smooth gradient from one end of the pipe to the other.) Mathematically, the problem is to solve a 1D differential equation representing heat diffusion:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{4.2}$$

The approach used is to discretize the problem space (representing U by a one-dimensional array and computing values for a sequence of discrete time steps). We will output values for each time step as they are computed, so we need only save values for U for two time steps; we will call these arrays `uk` (U at the timestep k) and `ukp1` (U at timestep $k+1$). At each time step, we then need to compute for each point in array `ukp1` the following:

```
ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
```

Variables `dt` and `dx` represent the intervals between discrete time steps and between discrete points, respectively.

Observe that what is being computed is a new value for variable `ukp1` at each point, based on data at that point and its left and right neighbors.

We can begin to design a parallel algorithm for this problem by decomposing the arrays `uk` and `ukp1` into contiguous subarrays (the chunks described earlier). These chunks can be operated on concurrently, giving us exploitable concurrency. Notice that we have a situation in which some elements can be updated using only data from within the chunk, while others require data from neighboring chunks, as illustrated by Fig. 4.8.

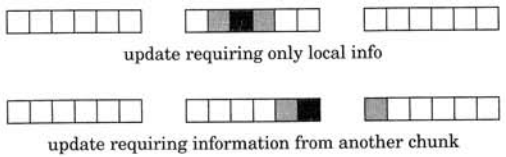

Figure 4.8: Data dependencies in the heat-equation problem. Solid boxes indicate the element being updated; shaded boxes the elements containing needed data.

Example: matrix-multiplication program. Consider the multiplication of two square matrices (that is, compute $C = A \cdot B$). As discussed in [FJL+88], the matrices can be decomposed into blocks. The summations in the definition of matrix multiplication are likewise organized into blocks, allowing us to write a blockwise matrix multiplication equation

$$C^{ij} = \sum_k A^{ik} \cdot B^{kj} \tag{4.3}$$

where at each step in the summation, we compute the matrix product $A^{ik} \cdot B^{kj}$ and add it to the running matrix sum.

This equation immediately implies a solution in terms of the *Geometric Decomposition* pattern; that is, one in which the algorithm is based on decomposing the data structure into chunks (square blocks here) that can be operated on concurrently.

To help visualize this algorithm more clearly, consider the case where we decompose all three matrices into square blocks with each task "owning" corresponding blocks of A, B, and C. Each task will run through the sum over k to compute its block of C, with tasks receiving blocks from other tasks as needed. In Fig. 4.9, we illustrate two steps in this process showing a block being updated (the solid block) and the matrix blocks required at two different steps (the shaded blocks), where blocks of the A matrix are passed across a row and blocks of the B matrix are passed around a column.

Forces

- To exploit the potential concurrency in the problem, we must assign chunks of the decomposed data structure to UEs. Ideally, we want to do this in a way that is simple, portable, scalable, and efficient. As noted in Sec. 4.1, however, these goals may conflict. A key consideration is balancing the load, that is, ensuring that all UEs have roughly the same amount of work to do.
- We must also ensure that the data required for the update of each chunk is present when needed. This problem is somewhat analogous to the problem

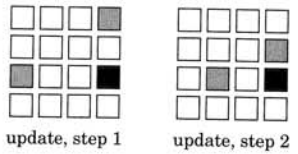

Figure 4.9: Data dependencies in the matrix-multiplication problem. Solid boxes indicate the "chunk" being updated (C); shaded boxes indicate the chunks of A (row) and B (column) required to update C at each of the two steps.

of managing data dependencies in the *Task Parallelism* pattern, and again the design must keep in mind the sometimes-conflicting goals of simplicity, portability, scalability, and efficiency.

Solution

Designs for problems that fit this pattern involve the following key elements: partitioning the global data structure into substructures or "chunks" (the data decomposition), ensuring that each task has access to all the data it needs to perform the update operation for its chunk (the exchange operation), updating the chunks (the update operation), and mapping chunks to UEs in a way that gives good performance (the data distribution and task schedule).

Data decomposition. The granularity of the data decomposition has a significant impact on the efficiency of the program. In a coarse-grained decomposition, there are a smaller number of large chunks. This results in a smaller number of large messages, which can greatly reduce communication overhead. A fine-grained decomposition, on the other hand, results in a larger number of smaller chunks, in many cases leading to many more chunks than PEs. This results in a larger number of smaller messages (and hence increases communication overhead), but it greatly facilitates load balancing.

Although it might be possible in some cases to mathematically derive an optimum granularity for the data decomposition, programmers usually experiment with a range of chunk sizes to empirically determine the best size for a given system. This depends, of course, on the computational performance of the PEs and on the performance characteristics of the communication network. Therefore, the program should be implemented so that the granularity is controlled by parameters that can be easily changed at compile or runtime.

The shape of the chunks can also affect the amount of communication needed between tasks. Often, the data to share between tasks is limited to the boundaries of the chunks. In this case, the amount of shared information scales with the surface area of the chunks. Because the computation scales with the number of points within a chunk, it scales as the volume of the region. This surface-to-volume effect can be exploited to maximize the ratio of computation to communication. Therefore, higher-dimensional decompositions are usually preferred. For example, consider two different decompositions of an N by N matrix into four chunks. In one case, we decompose the problem into four column chunks of size N by $N/4$. In the second case, we decompose the problem into four square chunks of size $N/2$ by $N/2$. For the column block decomposition, the surface area is $2N + 2(N/4)$ or $5N/2$. For the square chunk case, the surface area is $4(N/2)$ or $2N$. Hence, the total amount of data that must be exchanged is less for the square chunk decomposition.

In some cases, the preferred shape of the decomposition can be dictated by other concerns. It may be the case, for example, that existing sequential code can be more easily reused with a lower-dimensional decomposition, and the potential increase in performance is not worth the effort of reworking the code. Also, an instance of this pattern can be used as a sequential step in a larger computation. If

the decomposition used in an adjacent step differs from the optimal one for this pattern in isolation, it may or may not be worthwhile to redistribute the data for this step. This is especially an issue in distributed-memory systems where redistributing the data can require significant communication that will delay the computation. Therefore, data decomposition decisions must take into account the capability to reuse sequential code and the need to interface with other steps in the computation. Notice that these considerations might lead to a decomposition that would be suboptimal under other circumstances.

Communication can often be more effectively managed by replicating the nonlocal data needed to update the data in a chunk. For example, if the data structure is an array representing the points on a mesh and the update operation uses a local neighborhood of points on the mesh, a common communication-management technique is to surround the data structure for the block with a *ghost boundary* to contain duplicates of data at the boundaries of neighboring blocks. So now each chunk has two parts: a primary copy owned by the UE (that will be updated directly) and zero or more ghost copies (also referred to as shadow copies). These ghost copies provide two benefits. First, their use may consolidate communication into potentially fewer, larger messages. On latency-sensitive networks, this can greatly reduce communication overhead. Second, communication of the ghost copies can be *overlapped* (that is, it can be done concurrently) with the update of parts of the array that don't depend on data within the ghost copy. In essence, this hides the communication cost behind useful computation, thereby reducing the observed communication overhead.

For example, in the case of the mesh-computation example discussed earlier, each of the chunks would be extended by one cell on each side. These extra cells would be used as ghost copies of the cells on the boundaries of the chunks. Fig. 4.10 illustrates this scheme.

The exchange operation. A key factor in using this pattern correctly is ensuring that nonlocal data required for the update operation is obtained before it is needed.

If all the data needed is present before the beginning of the update operation, the simplest approach is to perform the entire exchange before beginning the update, storing the required nonlocal data in a local data structure designed for that purpose (for example, the ghost boundary in a mesh computation). This approach is relatively straightforward to implement using either copying or message passing.

More sophisticated approaches in which computation and communication overlap are also possible. Such approaches are necessary if some data needed for the update is not initially available, and may improve performance in other cases

Figure 4.10: A data distribution with ghost boundaries. Shaded cells are ghost copies; arrows point from primary copies to corresponding secondary copies.

as well. For example, in the example of a mesh computation, the exchange of ghost cells and the update of cells in the interior region (which do not depend on the ghost cells) can proceed concurrently. After the exchange is complete, the boundary layer (the values that do depend on the ghost cells) can be updated. On systems where communication and computation occur in parallel, the savings from such an approach can be significant. This is such a common feature of parallel algorithms that standard communication APIs (such as MPI) include whole classes of message-passing routines to overlap computation and communication. These are discussed in more detail in the MPI appendix.

The low-level details of how the exchange operation is implemented can have a large impact on efficiency. Programmers should seek out optimized implementations of communication patterns used in their programs. In many applications, for example, the collective communication routines in message-passing libraries such as MPI are useful. These have been carefully optimized using techniques beyond the ability of many parallel programmers (we discuss some of these in Sec. 6.4.2) and should be used whenever possible.

The update operation. Updating the data structure is done by executing the corresponding tasks (each responsible for the update of one chunk of the data structures) concurrently. If all the needed data is present at the beginning of the update operation, and if none of this data is modified during the course of the update, parallelization is easier and more likely to be efficient.

If the required exchange of information has been performed before beginning the update operation, the update itself is usually straightforward to implement—it is essentially identical to the analogous update in an equivalent sequential program, particularly if good choices have been made about how to represent nonlocal data.

If the exchange and update operations overlap, more care is needed to ensure that the update is performed correctly. If a system supports lightweight threads that are well integrated with the communication system, then overlap can be achieved via multithreading within a single task, with one thread computing while another handles communication. In this case, synchronization between the threads is required.

In some systems, for example MPI, nonblocking communication is supported by matching communication primitives: one to start the communication (without blocking), and the other (blocking) to complete the operation and use the results. For maximal overlap, communication should be started as soon as possible, and completed as late as possible. Sometimes, operations can be reordered to allow more overlap without changing the algorithm semantics.

Data distribution and task scheduling. The final step in designing a parallel algorithm for a problem that fits this pattern is deciding how to map the collection of tasks (each corresponding to the update of one chunk) to UEs. Each UE can then be said to "own" a collection of chunks and the data they contain. Thus, we have a two-tiered scheme for distributing data among UEs: partitioning the data into chunks and then assigning these chunks to UEs. This scheme is

flexible enough to represent a variety of popular schemes for distributing data among UEs.

In the simplest case, each task can be statically assigned to a separate UE; then all tasks can execute concurrently, and the intertask coordination needed to implement the exchange operation is straightforward. This approach is most appropriate when the computation times of the tasks are uniform and the exchange operation has been implemented to overlap communication and computation within each tasks.

The simple approach can lead to poor load balance in some situations, however. For example, consider a linear algebra problem in which elements of the matrix are successively eliminated as the computation proceeds. Early in the computation, all the rows and columns of the matrix have numerous elements to work with and decompositions based on assigning full rows or columns to UEs are effective. Later in the computation, however, rows or columns become sparse, the work per row becomes uneven, and the computational load becomes poorly balanced between UEs. The solution is to decompose the problem into many more chunks than there are UEs and to scatter them among the UEs with a cyclic or block-cyclic distribution. (Cyclic and block-cyclic distributions are discussed in the *Distributed Array* pattern.) Then, as chunks become sparse, there are (with high probability) other nonsparse chunks for any given UE to work on, and the load becomes well balanced. A rule of thumb is that one needs around ten times as many tasks as UEs for this approach to work well.

It is also possible to use dynamic load-balancing algorithms to periodically redistribute the chunks among the UEs to improve the load balance. These incur overhead that must be traded off against the improvement likely to occur from the improved load balance and increased implementation costs. In addition, the resulting program is more complex than those that use one of the static methods. Generally, one should consider the (static) cyclic allocation strategy first.

Program structure. The overall program structure for applications of this pattern will normally use either the *Loop Parallelism* pattern or the *SPMD* pattern, with the choice determined largely by the target platform. These patterns are described in the *Supporting Structures* design space.

Examples

We include two examples with this pattern: a mesh computation and matrix multiplication. The challenges in working with the *Geometric Decomposition* pattern are best appreciated in the low-level details of the resulting programs. Therefore, even though the techniques used in these programs are not fully developed until much later in the book, we provide full programs in this section rather than high-level descriptions of the solutions.

Mesh computation. This problem is described in the Context section of this pattern. Fig. 4.11 presents a simple sequential version of a program (some details omitted) that solves the 1D heat-diffusion problem. The program associated with

```
#include <stdio.h>
#include <stdlib.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) {
  uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
  for (int i = 1; i < NX-1; ++i)
    uk[i] = 0.0;
  for (int i = 0; i < NX; ++i)
    ukp1[i] = uk[i];
}

void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk = malloc(sizeof(double) * NX);
  double *ukp1 = malloc(sizeof(double) * NX);
  double *temp;

  double dx = 1.0/NX;
  double dt = 0.5*dx*dx;

  initialize(uk, ukp1);

  for (int k = 0; k < NSTEPS; ++k) {

    /* compute new values */
    for (int i = 1; i < NX-1; ++i) {
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }

    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k);
  }
  return 0;
}
```

Figure 4.11: Sequential heat-diffusion program

this problem is straightforward, although one detail might need further explanation: After computing new values in `ukp1` at each step, conceptually what we want to do is copy them to `uk` for the next iteration. We avoid a time-consuming actual copy by making `uk` and `ukp1` pointers to their respective arrays and simply swapping them at the end of each step. This causes `uk` to point to the newly computed values and `ukp1` to point to the area to use for computing new values in the next iteration.

This program combines a top-level sequential control structure (the time-step loop) with an array-update operation, which can be parallelized using the *Geometric Decomposition* pattern. We show parallel implementations of this program using OpenMP and MPI.

OpenMP solution. A particularly simple version of the program using OpenMP and the *Loop Parallelism* pattern is shown in Fig. 4.12. Because OpenMP is a shared-memory programming model, there is no need to explicitly partition and distribute the two key arrays (`uk` and `ukp1`). The creation of the threads and distribution of the work among the threads are accomplished with the `parallel for` directive.

```
#pragma parallel for schedule(static)
```

The `schedule(static)` clause decomposes the iterations of the parallel loop into one contiguous block per thread with each block being approximately the same size. This schedule is important for *Loop Parallelism* programs implementing the *Geometric Decomposition* pattern. Good performance for most *Geometric Decomposition* problems (and mesh programs in particular) requires that the data in the processor's cache be used many times before it is displaced by data from new cache lines. Using large blocks of contiguous loop iterations increases the chance that multiple values fetched in a cache line will be utilized and that subsequent loop iterations are likely to find at least some of the required data in cache.

The last detail to discuss for the program in Fig. 4.12 is the synchronization required to safely copy the pointers. It is essential that all of the threads complete their work before the pointers they manipulate are swapped in preparation for the next iteration. In this program, this synchronization happens automatically due to the implied barrier (see Sec. 6.3.2) at the end of the parallel loop.

The program in Fig. 4.12 works well with a small number of threads. When large numbers of threads are involved, however, the overhead incurred by placing the thread creation and destruction inside the loop over `k` would be prohibitive. We can reduce thread-management overhead by splitting the `parallel for` directive into separate `parallel` and `for` directives and moving the thread creation outside the loop over `k`. This approach is shown in Fig. 4.13. Because the whole `k` loop is now inside a parallel region, we must be more careful about how data is shared between threads. The `private` clause causes the loop indices `k` and `i` to be local to each thread. The pointers `uk` and `ukp1` are shared, however, so the swap operation must be protected. The easiest way to do this is to ensure that only one member of the team of threads does the swap. In OpenMP, this is most easily done by placing the update inside a `single` construct. As described in more detail in the OpenMP appendix, Appendix A, the first thread to encounter the construct will carry out the swap while the other threads wait at the end of the construct.

```
#include <stdio.h>
#include <stdlib.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) {
  uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
  for (int i = 1; i < NX-1; ++i)
    uk[i] = 0.0;
  for (int i = 0; i < NX; ++i)
    ukp1[i] = uk[i];
}

void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk = malloc(sizeof(double) * NX);
  double *ukp1 = malloc(sizeof(double) * NX);
  double *temp;

  double dx = 1.0/NX;
  double dt = 0.5*dx*dx;

  initialize(uk, ukp1);

  for (int k = 0; k < NSTEPS; ++k) {

    #pragma omp parallel for schedule(static)
    /* compute new values */
      for (int i = 1; i < NX-1; ++i) {
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
      }

    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k);
  }
  return 0;
}
```

Figure 4.12: Parallel heat-diffusion program using OpenMP

MPI solution. An MPI-based program for this example is shown in Figs. 4.14 and 4.15. The approach used in this program uses a data distribution with ghost cells and the *SPMD* pattern.

Each process is given a single chunk of the data domain of size `NX/NP`, where `NX` is the total size of the global data array and `NP` is the number of processes. For simplicity, we assume `NX` is evenly divided by `NP`.

The update of the chunk is straightforward and essentially identical to that from the sequential code. The length and greater complexity in this MPI program


```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]){/* NOT SHOWN */}
void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk = malloc(sizeof(double) * NX);
  double *ukp1 = malloc(sizeof(double) * NX);
  double *temp;
  int i,k;

  double dx = 1.0/NX;
  double dt = 0.5*dx*dx;

  #pragma omp parallel private (k, i)
  {
    initialize(uk, ukp1);

    for (k = 0; k < NSTEPS; ++k) {
      #pragma omp for schedule(static)
      for (i = 1; i < NX-1; ++i) {
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
      }
      /* "copy" ukp1 to uk by swapping pointers */
      #pragma omp single
      { temp = ukp1; ukp1 = uk; uk = temp; }
    }
  }
  return 0;
}
```

Figure 4.13: Parallel heat-diffusion program using OpenMP. This version has less thread-management overhead.

arises from two sources. First, the data initialization is more complex, because it must account for the data values at the edges of the first and last chunks. Second, message-passing routines are required inside the loop over `k` to exchange ghost cells.

The details of the message-passing functions can be found in the MPI appendix, Appendix B. Briefly, transmitting data consists of one process doing a *send* operation, specifying the buffer containing the data, and another process doing a *receive* operation, specifying the buffer into which the data should be placed. We need several different pairs of sends and receives because the process that owns the leftmost chunk of the array does not have a left neighbor it needs to communicate with, and similarly the process that owns the rightmost chunk does not have a right neighbor to communicate with.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[], int numPoints,
       int numProcs, int myID) {
  for (int i = 1; i <= numPoints; ++i)
     uk[i] = 0.0;
  /* left endpoint */
  if (myID == 0) uk[1] = LEFTVAL;
  /* right endpoint */
  if (myID == numProcs-1) uk[numPoints] = RIGHTVAL;
  /* copy values to ukp1 */
  for (int i = 1; i <= numPoints; ++i) ukp1[i] = uk[i];
}

void printValues(double uk[], int step, int numPoints, int myID)
  { /* NOT SHOWN */ }

int main(int argc, char *argv[]) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk, *ukp1, *temp;

  double dx = 1.0/NX; double dt = 0.5*dx*dx;

  int numProcs, myID, leftNbr, rightNbr, numPoints;
  MPI_Status status;

  /* MPI initialization */
  MPI_Init(&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID); //get own ID

  /* initialization of other variables */
  leftNbr = myID - 1; // ID of left "neighbor" process
  rightNbr = myID + 1; // ID of right "neighbor" process
  numPoints = (NX / numProcs);
  /* uk, ukp1 include a "ghost cell" at each end */
  uk = malloc(sizeof(double) * (numPoints+2));
  ukp1 = malloc(sizeof(double) * (numPoints+2));

  initialize(uk, ukp1, numPoints, numProcs, myID);
/* continued in next figure */
```

Figure 4.14: Parallel heat-diffusion program using MPI (continued in Fig. 4.15)

We could further modify the code in Figs. 4.14 and 4.15 to use nonblocking communication to overlap computation and communication, as discussed earlier in this pattern. The first part of the program is unchanged from our first mesh computation MPI program (that is, Fig. 4.14). The differences for this case are


```
/* continued from Figure 4.14 */

  for (int k = 0; k < NSTEPS; ++k) {

    /* exchange boundary information */
    if (myID != 0)
        MPI_Send(&uk[1], 1, MPI_DOUBLE, leftNbr, 0,
                                        MPI_COMM_WORLD);
    if (myID != numProcs-1)
        MPI_Send(&uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0,
                                        MPI_COMM_WORLD);
    if (myID != 0)
        MPI_Recv(&uk[0], 1, MPI_DOUBLE, leftNbr, 0,
                                        MPI_COMM_WORLD, &status);
    if (myID != numProcs-1)
        MPI_Recv(&uk[numPoints+1],1, MPI_DOUBLE, rightNbr, 0,
                                        MPI_COMM_WORLD, &status);

    /* compute new values for interior points */
    for (int i = 2; i < numPoints; ++i) {
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    /* compute new values for boundary points */
    if (myID != 0) {
        int i=1;
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    if (myID != numProcs-1) {
        int i=numPoints;
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }

    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k, numPoints, myID);
  }

  /* clean up and end */
  MPI_Finalize();
  return 0;
}
```

Figure 4.15: Parallel heat-diffusion program using MPI (continued from Fig. 4.14)

contained in the second part of the program containing the main computation loop. This code is shown in Fig. 4.16.

While the basic algorithm is the same, the communication is quite different. The *immediate-mode* communication routines, `MPI_Isend` and `MPI_Irecv`, are used to set up and then launch the communication events. These functions (described in more detail in the MPI appendix, Appendix B) return immediately. The update operations on the interior points can then take place because they don't depend on the results of the communication. We then call functions to wait until the communication is complete and update the edges of each UE's chunks using the results

```
/* continued */
  MPI_Request reqRecvL, reqRecvR, reqSendL, reqSendR; //needed for
                                                      // nonblocking I/O

  for (int k = 0; k < NSTEPS; ++k) {
    /* initiate communication to exchange boundary information */
    if (myID != 0) {
      MPI_Irecv(&uk[0], 1, MPI_DOUBLE, leftNbr, 0,
                MPI_COMM_WORLD, &reqRecvL);
      MPI_Isend(&uk[1], 1, MPI_DOUBLE, leftNbr, 0,
                   MPI_COMM_WORLD, &reqSendL);
    }
    if (myID != numProcs-1) {
      MPI_Irecv(&uk[numPoints+1],1, MPI_DOUBLE, rightNbr, 0,
                MPI_COMM_WORLD, &reqRecvR);
      MPI_Isend(&uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0,
                MPI_COMM_WORLD, &reqSendR);
    }
    /* compute new values for interior points */
    for (int i = 2; i < numPoints; ++i) {
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    /* wait for communication to complete */
    if (myID != 0) {
      MPI_Wait(&reqRecvL, &status); MPI_Wait(&reqSendL, &status);
    }
    if (myID != numProcs-1) {
      MPI_Wait(&reqRecvR, &status); MPI_Wait(&reqSendR, &status);
    }
    /* compute new values for boundary points */
    if (myID != 0) {
      int i=1;
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    if (myID != numProcs-1) {
      int i=numPoints;
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k, numPoints, myID);
  }
  /* clean up and end */
  MPI_Finalize();
  return 0;
}
```

Figure 4.16: Parallel heat-diffusion program using MPI with overlapping communication/computation (continued from Fig. 4.14)

of the communication events. In this case, the messages are small in size, so it is unlikely that this version of the program would be any faster than our first one. But it is easy to imagine cases where large, complex communication events would be involved and being able to do useful work while the messages move across the computer network would result in significantly greater performance.


```
#include <stdio.h>
#include <stdlib.h>
#define N 100
#define NB 4

#define blockstart(M,i,j,rows_per_blk,cols_per_blk,stride) \
  (M + ((i)*(rows_per_blk))*(stride) + (j)*(cols_per_blk))

int main(int argc, char *argv[]) {
  /* matrix dimensions */
  int dimN = N; int dimP = N; int dimM = N;

  /* block dimensions */
  int dimNb = dimN/NB; int dimPb = dimP/NB; int dimMb = dimM/NB;

  /* allocate memory for matrices */
  double *A = malloc(dimN*dimP*sizeof(double));
  double *B = malloc(dimP*dimM*sizeof(double));
  double *C = malloc(dimN*dimM*sizeof(double));

  /* Initialize matrices */

  initialize(A, B, dimN, dimP, dimM);

  /* Do the matrix multiplication */

  for (int ib=0; ib < NB; ++ib) {
    for (int jb=0; jb < NB; ++jb) {
      /* find block[ib][jb] of C */
      double * blockPtr = blockstart(C, ib, jb, dimNb, dimMb, dimM);
      /* clear block[ib][jb] of C (set all elements to zero) */
      matclear(blockPtr, dimNb, dimMb, dimM);
      for (int kb=0; kb < NB; ++kb) {
        /* compute product of block[ib][kb] of A and
           block[kb][jb] of B and add to block[ib][jb] of C */
        matmul_add(blockstart(A, ib, kb, dimNb, dimPb, dimP),
                   blockstart(B, kb, jb, dimPb, dimMb, dimM),
                   blockPtr, dimNb, dimPb, dimMb, dimP, dimM, dimM);
      }
    }
  }

  /* Code to print results not shown */

  return 0;
}
```

Figure 4.17: Sequential matrix multiplication

Matrix multiplication. The matrix multiplication problem is described in the Context section. Fig. 4.17 presents a simple sequential program to compute the desired result, based on decomposing the `N` by `N` matrix into `NB*NB` square blocks. The notation `block[i][j]` in comments indicates the (i,j)-th block as described earlier. To simplify the coding in C, we represent the matrices as 1D arrays (internally arranged in row-major order) and define a macro `blockstart` to find the top-left

```
/* Declarations, initializations, etc. not shown -- same as
   first version */

/* Do the multiply */

matclear(C, dimN, dimM, dimM); /* sets all elements to zero */

for (int kb=0; kb < NB; ++kb) {

  for (int ib=0; ib < NB; ++ib) {
    for (int jb=0; jb < NB; ++jb) {
      /* compute product of block[ib][kb] of A and
         block[kb][jb] of B and add to block[ib][jb] of C */
      matmul_add(blockstart(A, ib, kb, dimNb, dimPb, dimP),
                 blockstart(B, kb, jb, dimPb, dimMb, dimM),
                 blockstart(C, ib, jb, dimNb, dimMb, dimM),
                 dimNb, dimPb, dimMb, dimP, dimM, dimM);
    }
  }
}

/* Remaining code is the same as for the first version */
```

Figure 4.18: Sequential matrix multiplication, revised. We do not show the parts of the program that are not changed from the program in Fig. 4.17.

corner of a submatrix within one of these 1D arrays. We omit code for functions `initialize` (initialize matrices `A` and `B`), `printMatrix` (print a matrix's values), `matclear` (clear a matrix—set all values to zero), and `matmul_add` (compute the matrix product of the two input matrices and add it to the output matrix). Parameters to most of these functions include matrix dimensions, plus a stride that denotes the distance from the start of one row of the matrix to the start of the next and allows us to apply the functions to submatrices as well as to whole matrices.

We first observe that we can rearrange the loops without affecting the result of the computation, as shown in Fig. 4.18.

Observe that with this transformation, we have a program that combines a high-level sequential structure (the loop over `kb`) with a loop structure (the nested loops over `ib` and `jb`) that can be parallelized with the *Geometric Decomposition* pattern.

OpenMP solution. We can produce a parallel version of this program for a shared-memory environment by parallelizing the inner nested loops (over `ib` and/or `jb`) with OpenMP loop directives. As with the mesh example, it is important to keep thread-management overhead small, so once again the `parallel` directive should appear outside of the loop over `kb`. A `for` directive would then be placed prior to one of the inner loops. The issues raised by this algorithm and the resulting source code modifications are essentially the same as those arising from the mesh program example, so we do not show program source code here.

MPI solution. A parallel version of the matrix multiplication program using MPI is shown in Figs. 4.19 and 4.20. The natural approach with MPI is to use


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#define N 100

#define blockstart(M,i,j,rows_per_blk,cols_per_blk,stride) \
  (M + ((i)*(rows_per_blk))*(stride) + (j)*(cols_per_blk))

int main(int argc, char *argv[]) {
  /* matrix dimensions */
  int dimN = N; int dimP = N; int dimM = N;

  /* block dimensions */
  int dimNb, dimPb, dimMb;

  /* matrices */
  double *A, *B, *C;

  /* buffers for receiving sections of A, B from other processes */
  double *Abuffer, *Bbuffer;

  int numProcs, myID, myID_i, myID_j, NB;
  MPI_Status status;

  /* MPI initialization */
  MPI_Init(&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID);

  /* initialize other variables */
  NB = (int) sqrt((double) numProcs);
  myID_i = myID / NB;
  myID_j = myID % NB;
  dimNb = dimN/NB; dimPb = dimP/NB; dimMb = dimM/NB;
  A = malloc(dimNb*dimPb*sizeof(double));
  B = malloc(dimPb*dimMb*sizeof(double));
  C = malloc(dimNb*dimMb*sizeof(double));
  Abuffer = malloc(dimNb*dimPb*sizeof(double));
  Bbuffer = malloc(dimPb*dimMb*sizeof(double));

  /* Initialize matrices */
  initialize(A, B, dimNb, dimPb, dimMb, NB, myID_i, myID_j);

/* continued in next figure */
```

Figure 4.19: Parallel matrix multiplication with message passing (continued in Fig. 4.20)

the *SPMD* pattern with the *Geometric Decomposition* pattern. We will use the matrix multiplication algorithm described earlier.

The three matrices (`A`, `B`, and `C`) are decomposed into blocks. The UEs (processes in the case of MPI) involved in the computation are organized into a grid such that the indices of the matrix blocks map onto the coordinates of the processes (that is, matrix block `(i,j)` is associated with the process with row index `i` and column index `j`). For simplicity, we assume the number of processes `numProcs` is a perfect square and its square root evenly divides the order of the matrices (`N`).

```
/* continued from previous figure */

  /* Do the multiply */

  matclear(C, dimNb, dimMb, dimMb);
  for (int kb=0; kb < NB; ++kb) {

    if (myID_j == kb) {
      /* send A to other processes in the same "row" */
      for (int jb=0; jb < NB; ++jb) {
        if (jb != myID_j)
          MPI_Send(A, dimNb*dimPb, MPI_DOUBLE,
                   myID_i*NB + jb, 0, MPI_COMM_WORLD);
      }
      /* copy A to Abuffer */
      memcpy(Abuffer, A, dimNb*dimPb*sizeof(double));
    }
    else {
      MPI_Recv(Abuffer, dimNb*dimPb, MPI_DOUBLE,
               myID_i*NB + kb, 0, MPI_COMM_WORLD, &status);
    }
    if (myID_i == kb) {
      /* send B to other processes in the same "column" */
      for (int ib=0; ib < NB; ++ib) {
        if (ib != myID_i)
          MPI_Send(B, dimPb*dimMb, MPI_DOUBLE,
                   ib*NB + myID_j, 0, MPI_COMM_WORLD);
      }
      /* copy B to Bbuffer */
       memcpy(Bbuffer, B, dimPb*dimMb*sizeof(double));
    }
    else {
      MPI_Recv(Bbuffer, dimPb*dimMb, MPI_DOUBLE,
               kb*NB + myID_j, 0, MPI_COMM_WORLD, &status);
    }

    /* compute product of block[ib][kb] of A and
       block[kb][jb] of B and add to block[ib][jb] of C */
    matmul_add(Abuffer, Bbuffer, C,
               dimNb, dimPb, dimMb, dimPb, dimMb, dimMb);
  }
  /* Code to print results not shown */

  /* Clean up and end */
  MPI_Finalize();
  return 0;
}
```

Figure 4.20: Parallel matrix multiplication with message-passing (continued from Fig. 4.19)

Although the algorithm may seem complex at first, the overall idea is straightforward. The computation proceeds through a number of phases (the loop over `kb`). At each phase, the process whose row index equals the `kb` index sends its blocks of `A` across the row of processes. Likewise, the process whose column index equals `kb` sends its blocks of `B` along the column of processes. Following the communication operations, each process then multiplies the `A` and `B` blocks it received and sums

the result into its block of `C`. After `NB` phases, the block of the `C` matrix on each process will hold the final product.

These types of algorithms are very common when working with MPI. The key to understanding these algorithms is to think in terms of the set of processes, the data owned by each process, and how data from neighboring processes flows among the processes as the calculation unfolds. We revisit these issues in the *SPMD* and *Distributed Array* patterns as well as in the MPI appendix.

A great deal of research has been carried out on parallel matrix multiplication and related linear algebra algorithms. A more sophisticated approach, in which the blocks of `A` and `B` circulate among processes, arriving at each process just in time to be used, is given in [FJL+88].

Known uses. Most problems involving the solution of differential equations use the *Geometric Decomposition* pattern. A finite-differencing scheme directly maps onto this pattern. Another class of problems that use this pattern comes from computational linear algebra. The parallel routines in the ScaLAPACK [Sca, BCC+97] library are for the most part based on this pattern. These two classes of problems cover a large portion of all parallel applications in scientific computing.

Related Patterns

If the update required for each chunk can be done without data from other chunks, then this pattern reduces to the embarrassingly parallel algorithm described in the *Task Parallelism* pattern. As an example of such a computation, consider computing a 2D FFT (Fast Fourier Transform) by first applying a 1D FFT to each row of the matrix and then applying a 1D FFT to each column. Although the decomposition may appear data-based (by rows/by columns), in fact the computation consists of two instances of the *Task Parallelism* pattern.

If the data structure to be distributed is recursive in nature, then the *Divide and Conquer* or *Recursive Data* pattern may be applicable.

4.7 THE *RECURSIVE DATA* PATTERN

Problem

Suppose the problem involves an operation on a recursive data structure (such as a list, tree, or graph) that appears to require sequential processing. How can operations on these data structures be performed in parallel?

Context

Some problems with recursive data structures naturally use the divide-and-conquer strategy described in the *Divide and Conquer* pattern with its inherent potential for concurrency. Other operations on these data structures, however, seem to have little if any potential for concurrency because it appears that the only way to solve the problem is to sequentially move through the data structure, computing a result at one element before moving on to the next. Sometimes, however, it is possible

to reshape the operations in a way that a program can operate concurrently on all elements of the data structure.

An example from [J92] illustrates the situation: Suppose we have a forest of rooted directed trees (defined by specifying, for each node, its immediate ancestor, with a root node's ancestor being itself) and want to compute, for each node in the forest, the root of the tree containing that node. To do this in a sequential program, we would probably trace depth-first through each tree from its root to its leaf nodes; as we visit each node, we have the needed information about the corresponding root. Total running time of such a program for a forest of N nodes would be $O(N)$. There is some potential for concurrency (operating on subtrees concurrently), but there is no obvious way to operate on *all* elements concurrently, because it appears that we cannot find the root for a particular node without knowing its parent's root.

However, a rethinking of the problem exposes additional concurrency: We first define for each node a "successor", which initially will be its parent and ultimately will be the root of the tree to which the node belongs. We then calculate for each node its "successor's successor". For nodes one "hop" from the root, this calculation does not change the value of its successor (because a root's parent is itself). For nodes at least two "hops" away from a root, this calculation makes the node's successor its parent's parent. We repeat this calculation until it converges (that is, the values produced by one step are the same as those produced by the preceding step), at which point every node's successor is the desired value. Fig. 4.21 shows

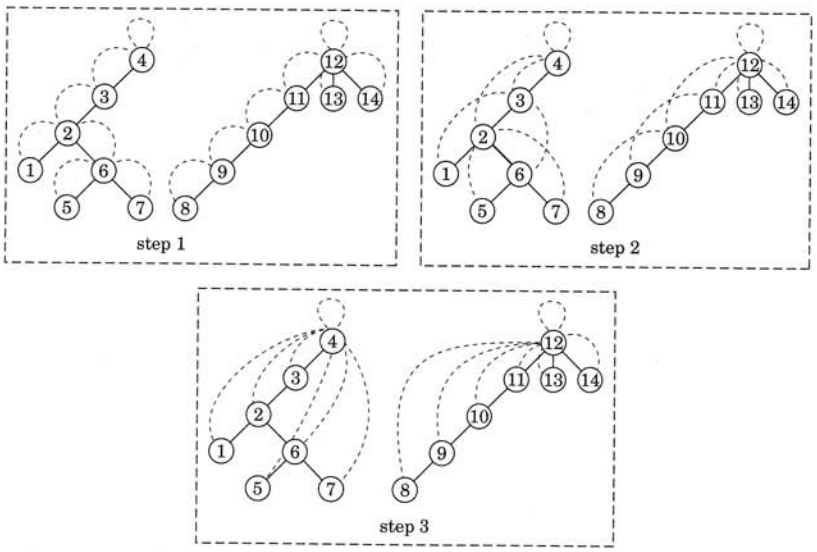

Figure 4.21: Finding roots in a forest. Solid lines represent the original parent-child relationships among nodes; dashed lines point from nodes to their successors.

an example requiring three steps to converge. At each step we can operate on all N nodes in the tree concurrently, and the algorithm converges in at most $\log N$ steps.

What we have done is transform the original sequential calculation (find roots for nodes one "hop" from a root, then find roots for nodes two "hops" from a root, etc.) into a calculation that computes a partial result (successor) for each node and then repeatedly combines these partial results, first with neighboring results, then with results from nodes two hops away, then with results from nodes four hops away, and so on. This strategy can be applied to other problems that at first appear unavoidably sequential; the Examples section presents other examples. This technique is sometimes referred to as *pointer jumping* or *recursive doubling*.

An interesting aspect of this restructuring is that the new algorithm involves substantially more total work than the original sequential one ($O(N \log N)$ versus $O(N)$), but the restructured algorithm contains potential concurrency that if fully exploited reduces total running time to $O(\log N)$ (versus $O(N)$). Most strategies and algorithms based on this pattern similarly trade off an increase in total work for a potential decrease in execution time. Notice also that the exploitable concurrency can be extremely fine-grained (as in the previous example), which may limit the situations in which this pattern yields an efficient algorithm. Nevertheless, the pattern can still serve as an inspiration for lateral thinking about how to parallelize problems that at first glance appear to be inherently sequential.

Forces

- Recasting the problem to transform an inherently sequential traversal of the recursive data structure into one that allows all elements to be operated upon concurrently does so at the cost of increasing the total work of the computation. This must be balanced against the improved performance available from running in parallel.
- This recasting may be difficult to achieve (because it requires looking at the original problem from an unusual perspective) and may lead to a design that is difficult to understand and maintain.
- Whether the concurrency exposed by this pattern can be effectively exploited to improve performance depends on how computationally expensive the operation is and on the cost of communication relative to computation on the target parallel computer system.

Solution

The most challenging part of applying this pattern is restructuring the operations over a recursive data structure into a form that exposes additional concurrency. General guidelines are difficult to construct, but the key ideas should be clear from the examples provided with this pattern.

After the concurrency has been exposed, it is not always the case that this concurrency can be effectively exploited to speed up the solution of a problem. This depends on a number of factors including how much work is involved as each

element of the recursive data structure is updated and on the characteristics of the target parallel computer.

Data decomposition. In this pattern, the recursive data structure is completely decomposed into individual elements and each element is assigned to a separate UE. Ideally each UE would be assigned to a different PE, but it is also possible to assign multiple UEs to each PE. If the number of UEs per PE is too large, however, the overall performance will be poor because there will not be enough concurrency to overcome the increase in the total amount of work.

For example, consider the root-finding problem described earlier. We'll ignore overhead in our computations. If $N = 1024$ and t is the time to perform one step for one data element, then the running time of a sequential algorithm will be about $1024t$. If each UE is assigned its own PE, then the running time of the parallel algorithm will be around $(\log N)t$ or $10t$. If only two PEs are available for the parallel algorithm, however, then all $N \log N$ or 10240 computation steps must be performed on the two PEs, and the execution time will be at least $5120t$, considerably more than the sequential algorithm.

Structure. Typically the result of applying this pattern is an algorithm whose top-level structure is a sequential composition in the form of a loop, in which each iteration can be described as "perform this operation simultaneously on all (or selected) elements of the recursive data structure". Typical operations include "replace each element's successor with its successor's successor" (as in the example in the Context section) and "replace a value held at this element with the sum of the current value and the value of the predecessor's element."

Synchronization. Algorithms that fit this pattern are described in terms of *simultaneously* updating all elements of the data structure. Some target platforms (for example, SIMD architectures such as the early Connection Machines) make this trivial to accomplish by assigning each data element to a separate PE (possibly a logical PE) and executing instructions in a lockstep fashion at each PE. MIMD platforms with the right supporting programming environments (for example, High Performance Fortran [HPF97]) provide similar semantics.

If the target platform doesn't provide the required synchronization implicitly, it will be necessary to introduce the synchronization explicitly. For example, if the operation performed during a loop iteration contains the assignment

```
next[k] = next[next[k]]
```

then the parallel algorithm must ensure that `next[k]` is not updated before other UEs that need its value for their computation have received it. One common technique is to introduce a new variable, say `next2`, at each element. Even-numbered iterations then read `next` but update `next2`, while odd-numbered iterations read

`next2` and update `next`. The necessary synchronization is accomplished by placing a barrier (as described in the *Implementation Mechanisms* design space) between each successive pair of iterations. Notice that this can substantially increase the overhead associated with the parallel algorithm, which can overwhelm any speedup derived from the additional concurrency. This is most likely to be a factor if the calculation required for each element is trivial (which, alas, for many of the examples it is).

If there are fewer PEs than data elements, the program designer must decide whether to assign each data element to a UE and assign multiple UEs to each PE (thereby simulating some of the parallelism) or whether to assign multiple data elements to each UE and process them serially. The latter is less straightforward (requiring an approach similar to that sketched previously, in which variables involved in the simultaneous update are duplicated), but can be more efficient.

Examples

Partial sums of a linked list. In this example, adopted from Hillis and Steele [HS86], the problem is to compute the prefix sums of all the elements in a linked list in which each element contains a value x. In other words, after the computation is complete, the first element will contain x_0, the second will contain $x_0 + x_1$, the third $x_0 + x_1 + x_2$, etc.

Fig. 4.22 shows pseudocode for the basic algorithm. Fig. 4.23 shows the evolution of the computation where x_i is the initial value of the $(i + 1)$-th element in the list.

This example can be generalized by replacing addition with any associative operator and is sometime known as a *prefix scan*. It can be used in a variety of situations, including solving various types of recurrence relations.

Known uses. Algorithms developed with this pattern are a type of *data parallel* algorithm. They are widely used on SIMD platforms and to a lesser extent in languages such as High Performance Fortran [HPF97]. These platforms support the fine-grained concurrency required for the pattern and handle synchronization

```
for all k in parallel
{
    temp[k] = next[k];
    while temp[k] != null
    {
        x[temp[k]] = x[k] + x[temp[k]];
        temp[k] = temp[temp[k]];
    }
}
```

Figure 4.22: Pseudocode for finding partial sums of a list

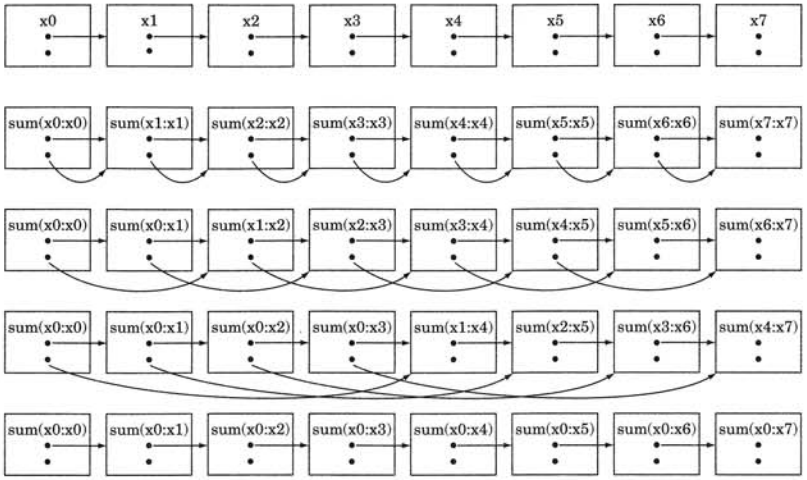

Figure 4.23: Steps in finding partial sums of a list. Straight arrows represent links between elements; curved arrows indicate additions.

automatically because every computation step (logically if not physically) occurs in lockstep on all the processors. Hillis and Steele [HS86] describe several interesting applications of this pattern, including finding the end of a linked list, computing all partial sums of a linked list, region labeling in two-dimensional images, and parsing.

In combinatorial optimization, problems involving traversing all nodes in a graph or tree can often be solved with this pattern by first finding an ordering on the nodes to create a list. Euler tours and ear decomposition [EG88] are well-known techniques to compute this ordering.

JáJá [J92] also describes several applications of this pattern: finding the roots of trees in a forest of rooted directed trees, computing partial sums on a set of rooted directed trees (similar to the preceding example with linked lists), and list-ranking (determining for each element of the list its distance from the start/end of the list).

Related Patterns

With respect to the actual concurrency, this pattern is very much like the *Geometric Decomposition* pattern, a difference being that in this pattern the data structure containing the elements to be operated on concurrently is recursive (at least conceptually). What makes it different is the emphasis on fundamentally rethinking the problem to expose fine-grained concurrency.

4.8 THE *PIPELINE* PATTERN

Problem

Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can the potential concurrency be exploited?

Context

An assembly line is a good analogy for this pattern. Suppose we want to manufacture a number of cars. The manufacturing process can be broken down into a sequence of operations each of which adds some component, say the engine or the windshield, to the car. An assembly line (pipeline) assigns a component to each worker. As each car moves down the assembly line, each worker installs the same component over and over on a succession of cars. After the pipeline is full (and until it starts to empty) the workers can all be busy simultaneously, all performing their operations on the cars that are currently at their stations.

Examples of pipelines are found at many levels of granularity in computer systems, including the CPU hardware itself.

- **Instruction pipeline in modern CPUs.** The stages (fetch instruction, decode, execute, etc.) are done in a pipelined fashion; while one instruction is being decoded, its predecessor is being executed and its successor is being fetched.

- **Vector processing (loop-level pipelining).** Specialized hardware in some supercomputers allows operations on vectors to be performed in a pipelined fashion. Typically, a compiler is expected to recognize that a loop such as

```
for(i = 0; i < N; i++) { a[i] = b[i] + c[i]; }
```

  can be vectorized in a way that the special hardware can exploit. After a short startup, one `a[i]` value will be generated each clock cycle.

- **Algorithm-level pipelining.** Many algorithms can be formulated as recurrence relations and implemented using a pipeline or its higher-dimensional generalization, a systolic array. Such implementations often exploit specialized hardware for performance reasons.

- **Signal processing.** Passing a stream of real-time sensor data through a sequence of filters can be modeled as a pipeline, with each filter corresponding to a stage in the pipeline.

- **Graphics.** Processing a sequence of images by applying the same sequence of operations to each image can be modeled as a pipeline, with each operation

corresponding to a pipeline stage. Some stages may be implemented by specialized hardware.

- **Shell programs in UNIX.** For example, the shell command

```
cat sampleFile | grep "word" | wc
```

creates a three-stage pipeline, with one process for each command (`cat`, `grep`, and `wc`).

These examples and the assembly-line analogy have several aspects in common. All involve applying a sequence of operations (in the assembly line case it is installing the engine, installing the windshield, etc.) to each element in a sequence of data elements (in the assembly line, the cars). Although there may be ordering constraints on the operations on a single data element (for example, it might be necessary to install the engine before installing the hood), it is possible to perform different operations on different data elements simultaneously (for example, one can install the engine on one car while installing the hood on another.)

The possibility of simultaneously performing different operations on different data elements is the potential concurrency this pattern exploits. In terms of the analysis described in the *Finding Concurrency* patterns, each task consists of repeatedly applying an operation to a data element (analogous to an assembly-line worker installing a component), and the dependencies among tasks are ordering constraints enforcing the order in which operations must be performed on each data element (analogous to installing the engine before the hood).

Forces

- A good solution should make it simple to express the ordering constraints. The ordering constraints in this problem are simple and regular and lend themselves to being expressed in terms of data flowing through a pipeline.
- The target platform can include special-purpose hardware that can perform some of the desired operations.
- In some applications, future additions, modifications, or reordering of the stages in the pipeline are expected.
- In some applications, occasional items in the input sequence can contain errors that prevent their processing.

Solution

The key idea of this pattern is captured by the assembly-line analogy, namely that the potential concurrency can be exploited by assigning each operation (stage

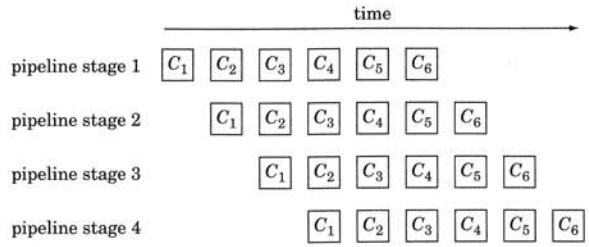

Figure 4.24: Operation of a pipeline. Each pipeline stage i computes the i-th step of the computation.

of the pipeline) to a different worker and having them work simultaneously, with the data elements passing from one worker to the next as operations are completed. In parallel-programming terms, the idea is to assign each task (stage of the pipeline) to a UE and provide a mechanism whereby each stage of the pipeline can send data elements to the next stage. This strategy is probably the most straightforward way to deal with this type of ordering constraints. It allows the application to take advantage of special-purpose hardware by appropriate mapping of pipeline stages to PEs and provides a reasonable mechanism for handling errors, described later. It also is likely to yield a modular design that can later be extended or modified.

Before going further, it may help to illustrate how the pipeline is supposed to operate. Let C_i represent a multistep computation on data element i. $C_i(j)$ is the jth step of the computation. The idea is to map computation steps to pipeline stages so that each stage of the pipeline computes one step. Initially, the first stage of the pipeline performs $C_1(1)$. After that completes, the second stage of the pipeline receives the first data item and computes $C_1(2)$ while the first stage computes the first step of the second item, $C_2(1)$. Next, the third stage computes $C_1(3)$, while the second stage computes $C_2(2)$ and the first stage $C_3(1)$. Fig. 4.24 illustrates how this works for a pipeline consisting of four stages. Notice that concurrency is initially limited and some resources remain idle until all the stages are occupied with useful work. This is referred to as *filling the pipeline*. At the end of the computation (*draining the pipeline*), again there is limited concurrency and idle resources as the final item works its way through the pipeline. We want the time spent filling or draining the pipeline to be small compared to the total time of the computation. This will be the case if the number of stages is small compared to the number of items to be processed. Notice also that overall throughput/efficiency is maximized if the time taken to process a data element is roughly the same for each stage.

This idea can be extended to include situations more general than a completely linear pipeline. For example, Fig. 4.25 illustrates two pipelines, each with four stages. In the second pipeline, the third stage consists of two operations that can be performed concurrently.

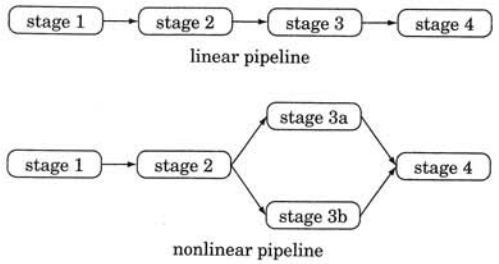

Figure 4.25: Example pipelines

Defining the stages of the pipeline. Normally each pipeline stage will correspond to one task. Fig. 4.26 shows the basic structure of each stage.

If the number of data elements to be processed is known in advance, then each stage can count the number of elements and terminate when these have been processed. Alternatively, a sentinel indicating termination may be sent through the pipeline.

It is worthwhile to consider at this point some factors that affect performance.

- The amount of concurrency in a full pipeline is limited by the number of stages. Thus, a larger number of stages allows more concurrency. However, the data sequence must be transferred between the stages, introducing overhead to the calculation. Thus, we need to organize the computation into stages such that the work done by a stage is large compared to the communication overhead. What is "large enough" is highly dependent on the particular architecture. Specialized hardware (such as vector processors) allows very fine-grained parallelism.

- The pattern works better if the operations performed by the various stages of the pipeline are all about equally computationally intensive. If the stages in the pipeline vary widely in computational effort, the slowest stage creates a bottleneck for the aggregate throughput.

```
initialize
while (more data)
{
    receive data element from previous stage
    perform operation on data element
    send data element to next stage
}
finalize
```

Figure 4.26: Basic structure of a pipeline stage

- The pattern works better if the time required to fill and drain the pipeline is small compared to the overall running time. This time is influenced by the number of stages (more stages means more fill/drain time).

Therefore, it is worthwhile to consider whether the original decomposition into tasks should be revisited at this point, possibly combining lightly-loaded adjacent pipeline stages into a single stage, or decomposing a heavily-loaded stage into multiple stages.

It may also be worthwhile to parallelize a heavily-loaded stage using one of the other *Algorithm Structure* patterns. For example, if the pipeline is processing a sequence of images, it is often the case that each stage can be parallelized using the *Task Parallelism* pattern.

Structuring the computation. We also need a way to structure the overall computation. One possibility is to use the *SPMD* pattern (described in the *Supporting Structures* design space) and use each UE's ID to select an option in a `case` or `switch` statement, with each case corresponding to a stage of the pipeline.

To increase modularity, object-oriented frameworks can be developed that allow stages to be represented by objects or procedures that can easily be "plugged in" to the pipeline. Such frameworks are not difficult to construct using standard OOP techniques, and several are available as commercial or freely available products.

Representing the dataflow among pipeline elements. How dataflow between pipeline elements is represented depends on the target platform.

In a message-passing environment, the most natural approach is to assign one process to each operation (stage of the pipeline) and implement each connection between successive stages of the pipeline as a sequence of messages between the corresponding processes. Because the stages are hardly ever perfectly synchronized, and the amount of work carried out at different stages almost always varies, this flow of data between pipeline stages must usually be both buffered and ordered. Most message-passing environments (e.g., MPI) make this easy to do. If the cost of sending individual messages is high, it may be worthwhile to consider sending multiple data elements in each message; this reduces total communication cost at the expense of increasing the time needed to fill the pipeline.

If a message-passing programming environment is not a good fit with the target platform, the stages of the pipeline can be connected explicitly with buffered channels. Such a buffered channel can be implemented as a queue shared between the sending and receiving tasks, using the *Shared Queue* pattern.

If the individual stages are themselves implemented as parallel programs, then more sophisticated approaches may be called for, especially if some sort of data redistribution needs to be performed between the stages. This might be the case if, for example, the data needs to be partitioned along a different dimension or partitioned into a different number of subsets in the same dimension. For example, an application might include one stage in which each data element is partitioned into three subsets and another stage in which it is partitioned into four subsets.

The simplest ways to handle such situations are to aggregate and disaggregate data elements between stages. One approach would be to have only one task in each stage communicate with tasks in other stages; this task would then be responsible for interacting with the other tasks in its stage to distribute input data elements and collect output data elements. Another approach would be to introduce additional pipeline stages to perform aggregation/disaggregation operations. Either of these approaches, however, involves a fair amount of communication. It may be preferable to have the earlier stage "know" about the needs of its successor and communicate with each task receiving part of its data directly rather than aggregating the data at one stage and then disaggregating at the next. This approach improves performance at the cost of reduced simplicity, modularity, and flexibility.

Less traditionally, networked file systems have been used for communication between stages in a pipeline running in a workstation cluster. The data is written to a file by one stage and read from the file by its successor. Network file systems are usually mature and fairly well optimized, and they provide for the visibility of the file at all PEs as well as mechanisms for concurrency control. Higher-level abstractions such as tuple spaces and blackboards implemented over networked file systems can also be used. File-system-based solutions are appropriate in large-grained applications in which the time needed to process the data at each stage is large compared with the time to access the file system.

Handling errors. For some applications, it might be necessary to gracefully handle error conditions. One solution is to create a separate task to handle errors. Each stage of the regular pipeline sends to this task any data elements it cannot process along with error information and then continues with the next item in the pipeline. The error task deals with the faulty data elements appropriately.

Processor allocation and task scheduling. The simplest approach is to allocate one PE to each stage of the pipeline. This gives good load balance if the PEs are similar and the amount of work needed to process a data element is roughly the same for each stage. If the stages have different requirements (for example, one is meant to be run on special-purpose hardware), this should be taken into consideration in assigning stages to PEs.

If there are fewer PEs than pipeline stages, then multiple stages must be assigned to the same PE, preferably in a way that improves or at least does not much reduce overall performance. Stages that do not share many resources can be allocated to the same PE; for example, a stage that writes to a disk and a stage that involves primarily CPU computation might be good candidates to share a PE. If the amount of work to process a data element varies among stages, stages involving less work may be allocated to the same PE, thereby possibly improving load balance. Assigning adjacent stages to the same PE can reduce communication costs. It might also be worthwhile to consider combining adjacent stages of the pipeline into a single stage.

If there are more PEs than pipeline stages, it is worthwhile to consider parallelizing one or more of the pipeline stages using an appropriate *Algorithm Structure* pattern, as discussed previously, and allocating more than one PE to the parallelized stage(s). This is particularly effective if the parallelized stage was previously a bottleneck (taking more time than the other stages and thereby dragging down overall performance).

Another way to make use of more PEs than pipeline stages, *if* there are no temporal constraints among the data items themselves (that is, it doesn't matter if, say, data item 3 is computed before data item 2), is to run multiple independent pipelines in parallel. This can be considered an instance of the *Task Parallelism* pattern. This will improve the throughput of the overall calculation, but does not significantly improve the latency, however, since it still takes the same amount of time for a data element to traverse the pipeline.

Throughput and latency. There are few more factors to keep in mind when evaluating whether a given design will produce acceptable performance.

In many situations where the *Pipeline* pattern is used, the performance measure of interest is the throughput, the number of data items per time unit that can be processed after the pipeline is already full. For example, if the output of the pipeline is a sequence of rendered images to be viewed as an animation, then the pipeline must have sufficient throughput (number of items processed per time unit) to generate the images at the required frame rate.

In another situation, the input might be generated from real-time sampling of sensor data. In this case, there might be constraints on both the throughput (the pipeline should be able to handle all the data as it comes in without backing up the input queue and possibly losing data) and the latency (the amount of time between the generation of an input and the completion of processing of that input). In this case, it might be desirable to minimize latency subject to a constraint that the throughput is sufficient to handle the incoming data.

Examples

Fourier-transform computations. A type of calculation widely used in signal processing involves performing the following computations repeatedly on different sets of data.

1. Perform a discrete Fourier transform (DFT) on a set of data.
2. Manipulate the result of the transform elementwise.
3. Perform an inverse DFT on the result of the manipulation.

Examples of such calculations include convolution, correlation, and filtering operations ([PTV93]).

A calculation of this form can easily be performed by a three-stage pipeline.

- The first stage of the pipeline performs the initial Fourier transform; it repeatedly obtains one set of input data, performs the transform, and passes the result to the second stage of the pipeline.

- The second stage of the pipeline performs the desired elementwise manipulation; it repeatedly obtains a partial result (of applying the initial Fourier transform to an input set of data) from the first stage of the pipeline, performs its manipulation, and passes the result to the third stage of the pipeline. This stage can often itself be parallelized using one of the other *Algorithm Structure* patterns.
- The third stage of the pipeline performs the final inverse Fourier transform; it repeatedly obtains a partial result (of applying the initial Fourier transform and then the elementwise manipulation to an input set of data) from the second stage of the pipeline, performs the inverse Fourier transform, and outputs the result.

Each stage of the pipeline processes one set of data at a time. However, except during the initial filling of the pipeline, all stages of the pipeline can operate concurrently; while the first stage is processing the N-th set of data, the second stage is processing the $(N - 1)$-th set of data, and the third stage is processing the $(N - 2)$-th set of data.

Java pipeline framework. The figures for this example show a simple Java framework for pipelines and an example application.

The framework consists of a base class for pipeline stages, `PipelineStage`, shown in Fig. 4.27, and a base class for pipelines, `LinearPipeline`, shown in Fig. 4.28. Applications provide a subclass of `PipelineStage` for each desired stage, implementing its three abstract methods to indicate what the stage should do on the initial step, the computation steps, and the final step, and a subclass of `LinearPipeline` that implements its abstract methods to create an array containing the desired pipeline stages and the desired queues connecting the stages. For the queue connecting the stages, we use `LinkedBlockingQueue`, an implementation of the `BlockingQueue` interface. These classes are found in the `java.util.concurrent` package. These classes use generics to specify the type of objects the queue can hold. For example, `new LinkedBlockingQueue<String>` creates a `BlockingQueue` implemented by an underlying linked list that can hold `Strings`. The operations of interest are `put`, to add an object to the queue, and `take`, to remove an object. `take` blocks if the queue is empty. The class `CountDownLatch`, also found in the `java.util.concurrent` package, is a simple barrier that allows the program to print a message when it has terminated. Barriers in general, and `CountDownLatch` in particular, are discussed in the *Implementation Mechanisms* design space.

The remaining figures show code for an example application, a pipeline to sort integers. Fig. 4.29 is the required subclass of `LinearPipeline`, and Fig. 4.30 is the required subclass of `PipelineStage`. Additional pipeline stages to generate or read the input and to handle the output are not shown.

Known uses. Many applications in signal and image processing are implemented as pipelines.

The OPUS [SR98] system is a pipeline framework developed by the Space Telescope Science Institute originally to process telemetry data from the Hubble


```
import java.util.concurrent.*;

abstract class PipelineStage implements Runnable {

    BlockingQueue in;
    BlockingQueue out;
    CountDownLatch s;

    boolean done;

    //override to specify initialization step
    abstract void firstStep() throws Exception;
    //override to specify compute step
    abstract void step() throws Exception;
    //override to specify finalization step
    abstract void lastStep() throws Exception;


    void handleComputeException(Exception e)
    { e.printStackTrace(); }


    public void run()
    {
     try
       { firstStep();
         while(!done){ step();}
         lastStep();
       }
     catch(Exception e){handleComputeException(e);}
       finally {s.countDown();}
    }

    public void init(BlockingQueue in,
                     BlockingQueue out,
                     CountDownLatch s)
    { this.in = in; this.out = out; this.s = s;}

}
```

Figure 4.27: Base class for pipeline stages

Space Telescope and later employed in other applications. OPUS uses a blackboard architecture built on top of a network file system for interstage communication and includes monitoring tools and support for error handling.

Airborne surveillance radars use space-time adaptive processing (STAP) algorithms, which have been implemented as a parallel pipeline [CLW+00]. Each stage is itself a parallel algorithm, and the pipeline requires data redistribution between some of the stages.

Fx [GOS94], a parallelizing Fortran compiler based on HPF [HPF97], has been used to develop several example applications [DGO+94,SSOG93] that combine data parallelism (similar to the form of parallelism captured in the *Geometric Decomposition* pattern) and pipelining. For example, one application performs 2D Fourier transforms on a sequence of images via a two-stage pipeline (one stage for the row

```
import java.util.concurrent.*;

abstract class LinearPipeline {
    PipelineStage[] stages;
    BlockingQueue[] queues;
    int numStages;
    CountDownLatch s;

    //override method to create desired array of pipeline stage objects
    abstract PipelineStage[] getPipelineStages(String[] args);

    //override method to create desired array of BlockingQueues
    //element i of returned array contains queue between stages i and i+1
    abstract BlockingQueue[] getQueues(String[] args);

    LinearPipeline(String[] args)
    { stages = getPipelineStages(args);
      queues = getQueues(args);
      numStages = stages.length;
      s = new CountDownLatch(numStages);

      BlockingQueue in = null;
      BlockingQueue out = queues[0];
      for (int i = 0; i != numStages; i++)
      { stages[i].init(in,out,s);
        in = out;
        if (i < numStages-2) out = queues[i+1]; else out = null;
        }
    }

    public void start()
    { for (int i = 0; i != numStages; i++)
      { new Thread(stages[i]).start();
        }
    }
}
```

Figure 4.28: Base class for linear pipeline

transforms and one stage for the column transforms), with each stage being itself parallelized using data parallelism. The SIGPLAN paper ([SSOG93]) is especially interesting in that it presents performance figures comparing this approach with a straight data-parallelism approach.

[J92] presents some finer-grained applications of pipelining, including inserting a sequence of elements into a 2-3 tree and pipelined mergesort.

Related Patterns

This pattern is very similar to the *Pipes and Filters* pattern of [BMR+96]; the key difference is that this pattern explicitly discusses concurrency.

For applications in which there are no temporal dependencies between the data inputs, an alternative to this pattern is a design based on multiple sequential pipelines executing in parallel and using the *Task Parallelism* pattern.


```
import java.util.concurrent.*;

class SortingPipeline extends LinearPipeline {

  /*Creates an array of pipeline stages with the
  number of sorting stages given via args. Input
  and output stages are also included at the
  beginning and end of the array. Details are omitted.
  */
   PipelineStage[] getPipelineStages(String[] args)
    { //....
       return stages;
    }


    /* Creates an array of LinkedBlockingQueues to serve as
        communication channels between the stages. For this
        example, the first is restricted to hold Strings,
        the rest can hold Comparables. */
    BlockingQueue[] getQueues(String[] args)
    { BlockingQueue[] queues = new BlockingQueue[totalStages - 1];
      queues[0] = new LinkedBlockingQueue<String>();
      for (int i = 1; i!= totalStages -1; i++)
      { queues[i] = new LinkedBlockingQueue<Comparable>();}
      return queues;
    }


    SortingPipeline(String[] args)
    { super(args);
   }


    public static void main(String[] args)
      throws InterruptedException
    { //create pipeline
      LinearPipeline l = new SortingPipeline(args);
      l.start(); //start threads associated with stages
      l.s.await(); //terminate thread when all stages terminated.
      System.out.println("All threads terminated");
    }
}
```

Figure 4.29: Pipelined sort (main class)

At first glance, one might also expect that sequential solutions built using the *Chain of Responsibility* pattern [GHJV95] could be easily parallelized using the *Pipeline* pattern. In *Chain of Responsibility,* or *COR,* an "event" is passed along a chain of objects until one or more of the objects handle the event. This pattern is directly supported, for example, in the Java Servlet Specification[1] [SER] to enable filtering of HTTP requests. With Servlets, as well as other typical applications of *COR,* however, the reason for using the pattern is to support modular structuring of

[1] A Servlet is a Java program invoked by a Web server. The Java Servlets technology is included in the Java 2 Enterprise Edition platform for Web server applications.

```
class SortingStage extends PipelineStage
{
    Comparable val = null;
    Comparable input = null;

    void firstStep() throws InterruptedException
    { input = (Comparable)in.take();
      done = (input.equals("DONE"));
      val = input;
      return;
    }

    void step() throws InterruptedException
    { input = (Comparable)in.take();
      done = (input.equals("DONE"));
      if (!done)
      { if(val.compareTo(input)<0)
        { out.put(val); val = input; }
          else { out.put(input); }
      } else out.put(val);
    }

    void lastStep() throws InterruptedException
    { out.put("DONE"); }
}
```

Figure 4.30: Pipelined sort (sorting stage)

a program that will need to handle *independent* events in *different ways* depending on the event type. It may be that only one object in the chain will even handle the event. We expect that in most cases, the *Task Parallelism* pattern would be more appropriate than the *Pipeline* pattern. Indeed, Servlet container implementations already supporting multithreading to handle independent HTTP requests provide this solution for free.

The *Pipeline* pattern is similar to the *Event-Based Coordination* pattern in that both patterns apply to problems where it is natural to decompose the computation into a collection of semi-independent tasks. The difference is that the *Event-Based Coordination* pattern is irregular and asynchronous where the *Pipeline* pattern is regular and synchronous: In the *Pipeline* pattern, the semi-independent tasks represent the stages of the pipeline, the structure of the pipeline is static, and the interaction between successive stages is regular and loosely synchronous. In the *Event-Based Coordination* pattern, however, the tasks can interact in very irregular and asynchronous ways, and there is no requirement for a static structure.

4.9 THE *EVENT-BASED COORDINATION* PATTERN

Problem

Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data

between them which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?

Context

Some problems are most naturally represented as a collection of semi-independent entities interacting in an irregular way. What this means is perhaps clearest if we compare this pattern with the *Pipeline* pattern. In the *Pipeline* pattern, the entities form a linear pipeline, each entity interacts only with the entities to either side, the flow of data is one-way, and interaction occurs at fairly regular and predictable intervals. In the *Event-Based Coordination* pattern, in contrast, there is no restriction to a linear structure, no restriction that the flow of data be one-way, and the interaction takes place at irregular and sometimes unpredictable intervals.

As a real-world analogy, consider a newsroom, with reporters, editors, fact-checkers, and other employees collaborating on stories. As reporters finish stories, they send them to the appropriate editors; an editor can decide to send the story to a fact-checker (who would then eventually send it back) or back to the reporter for further revision. Each employee is a semi-independent entity, and their interaction (for example, a reporter sending a story to an editor) is irregular.

Many other examples can be found in the field of discrete-event simulation, that is, simulation of a physical system consisting of a collection of objects whose interaction is represented by a sequence of discrete "events". An example of such a system is the car-wash facility described in [Mis86]: The facility has two car-wash machines and an attendant. Cars arrive at random times at the attendant. Each car is directed by the attendant to a nonbusy car-wash machine if one exists, or queued if both machines are busy. Each car-wash machine processes one car at a time. The goal is to compute, for a given distribution or arrival times, the average time a car spends in the system (time being washed plus any time waiting for a nonbusy machine) and the average length of the queue that builds up at the attendant. The "events" in this system include cars arriving at the attendant, cars being directed to the car-wash machines, and cars leaving the machines. Fig. 4.31 sketches this example. Notice that it includes "source" and "sink" objects to make it easier to model cars arriving and leaving the facility. Notice also that the attendant must be notified when cars leave the car-wash machines so that it knows whether the machines are busy.

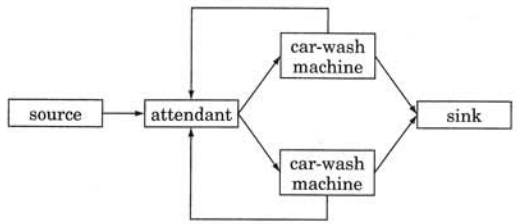

Figure 4.31: Discrete-event simulation of a car-wash facility. Arrows indicate the flow of events.

Also, it is sometimes desirable to compose existing, possibly sequential, program components that interact in possibly irregular ways into a parallel program without changing the internals of the components.

For problems such as this, it might make sense to base a parallel algorithm on defining a task (or a group of tightly coupled tasks) for each component, or in the case of discrete-event simulation, simulation entity. Interaction between these tasks is then based on the ordering constraints determined by the flow of data between them.

Forces

- A good solution should make it simple to express the ordering constraints, which can be numerous and irregular and even arise dynamically. It should also make it possible for as many activities as possible to be performed concurrently.
- Ordering constraints implied by the data dependencies can be expressed by encoding them into the program (for example, via sequential composition) or using shared variables, but neither approach leads to solutions that are simple, capable of expressing complex constraints, and easy to understand.

Solution

A good solution is based on expressing the data flow using abstractions called *events,* with each event having a task that generates it and a task that processes it. Because an event must be generated before it can be processed, events also define ordering constraints between the tasks. Computation within each task consists of processing events.

Defining the tasks. The basic structure of each task consists of receiving an event, processing it, and possibly generating events, as shown in Fig. 4.32.

If the program is being built from existing components, the task will serve as an instance of the *Facade* pattern [GHJV95] by providing a consistent event-based interface to the component.

The order in which tasks receive events must be consistent with the application's ordering constraints, as discussed later.

```
initialize
while(not done)
{
  receive event
  process event
  send events
}
finalize
```

Figure 4.32: Basic structure of a task in the *Event-Based Coordination* pattern

Representing event flow. To allow communication and computation to overlap, one generally needs a form of asynchronous communication of events in which a task can create (send) an event and then continue without waiting for the recipient to receive it. In a message-passing environment, an event can be represented by a message sent asynchronously from the task generating the event to the task that is to process it. In a shared-memory environment, a queue can be used to simulate message passing. Because each such queue will be accessed by more than one task, it must be implemented in a way that allows safe concurrent access, as described in the *Shared Queue* pattern. Other communication abstractions, such as tuple spaces as found in the Linda coordination language or JavaSpaces [FHA99], can also be used effectively with this pattern. Linda [CG91] is a simple language consisting of only six operations that read and write an associative (that is, content-addressable) shared memory called a *tuple space.* A tuple space is a conceptually shared repository for data containing objects called tuples that tasks use for communication in a distributed system.

Enforcing event ordering. The enforcement of ordering constraints may make it necessary for a task to process events in a different order from the order in which they are sent, or to wait to process an event until some other event from a given task has been received, so it is usually necessary to be able to look ahead in the queue or message buffer and remove elements out of order. For example, consider the situation in Fig. 4.33. Task 1 generates an event and sends it to task 2, which will process it, and also sends it to task 3, which is recording information about all events. Task 2 processes the event from task 1 and generates a new event, a copy of which is also sent to task 3. Suppose that the vagaries of the scheduling and underlying communication layer cause the event from task 2 to arrive before the event from task 1. Depending on what task 3 is doing with the events, this may or may not be problematic. If task 3 is simply tallying the number of events that occur, there is no problem. If task 3 is writing a log entry that should reflect the order in which events are handled, however, simply processing events in the order in which they arrive would in this case produce an incorrect result. If task 3 is controlling a gate, and the event from task 1 results in opening the gate and the event from task 2 in closing the gate, then the out-of-order messages could cause significant problems, and task 3 should not process the first event until after the event from task 1 has arrived and been processed.

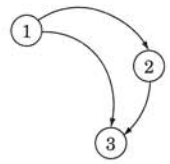

Figure 4.33: Event-based communication among three tasks. Task 2 generates its event in response to the event received from task 1. The two events sent to task 3 can arrive in either order.

In discrete-event simulations, a similar problem can occur because of the semantics of the application domain. An event arrives at a station (task) along with a simulation time when it should be scheduled. An event can arrive at a station before other events with earlier simulation times.

The first step is to determine whether, in a particular situation, out-of-order events can be a problem. There will be no problem if the "event" path is linear so that no out-of-order events will occur, or if, according to the application semantics, out-of-order events do not matter.

If out-of-order events may be a problem, then either an optimistic or pessimistic approach can be chosen. An optimistic approach requires the ability to roll back the effects of events that are mistakenly executed (including the effects of any new events that have been created by the out-of-order execution). In the area of distributed simulation, this approach is called time warp [Jef85]. Optimistic approaches are usually not feasible if an event causes interaction with the outside world. Pessimistic approaches ensure that the events are always executed in order at the expense of increased latency and communication overhead. Pessimistic approaches do not execute events until it can be guaranteed "safe" to do so. In the figure, for example, task 3 cannot process an event from task 2 until it "knows" that no earlier event will arrive from task 1 and vice versa. Providing task 3 with that knowledge may require introducing null events that contain no information useful for anything except the event ordering. Many implementations of pessimistic approaches are based on time stamps that are consistent with the causality in the system [Lam78].

Much research and development effort has gone into frameworks that take care of the details of event ordering in discrete-event simulation for both optimistic [RMC+98] and pessimistic approaches [CLL+99]. Similarly, middleware is available that handles event-ordering problems in process groups caused by the communication system. An example is the Ensemble system developed at Cornell [vRBH+98].

Avoiding deadlocks. It is possible for systems using this pattern to deadlock at the application level—for some reason the system arrives in a state where no task can proceed without first receiving an event from another task that will never arrive. This can happen because of a programming error; in the case of a simulation, it can also be caused by problems in the model that is being simulated. In the latter case, the developer must rethink the solution.

If pessimistic techniques are used to control the order in which events are processed, then deadlocks can occur when an event is available and actually could be processed, but is not processed because the event is not yet known to be safe. The deadlock can be broken by exchanging enough information that the event can be safely processed. This is a very significant problem as the overhead of dealing with deadlocks can cancel the benefits of parallelism and make the parallel algorithms slower than a sequential simulation. Approaches to dealing with this type of deadlock range from sending frequent enough "null messages" to avoid deadlocks altogether (at the cost of many extra messages) to using deadlock detection schemes to detect the presence of a deadlock and then resolve it (at the cost of possible significant idle time before the deadlock is detected and resolved). The approach of choice will depend on the frequency of deadlock. A middle-ground solution is to use timeouts instead of accurate deadlock detection, and is often the best approach.

Scheduling and processor allocation. The most straightforward approach is to allocate one task per PE and allow all the tasks to execute concurrently. If insufficient PEs are available to do this, then multiple tasks can be allocated to each PE. This should be done in a way that achieves good load balance. Load balancing is a difficult problem in this pattern due to its potentially irregular structure and possible dynamic nature. Some infrastructures that support this pattern allow task migration so that the load can be balanced dynamically at runtime.

Efficient communication of events. If the application is to perform well, the mechanism used to communicate events must be as efficient as is feasible. In a shared-memory environment, this means making sure the mechanism does not have the potential to become a bottleneck. In a message-passing environment, there are several efficiency considerations; for example, whether it makes sense to send many short messages between tasks or try to combine them. [YWC+96] and [WY95] describe some considerations and solutions.

Examples

Known uses. A number of discrete-event simulation applications use this pattern. The DPAT simulation used to analyze air traffic control systems [Wie01] is a successful simulation that uses optimistic techniques. It is implemented using the GTW (Georgia Tech Time Warp) System [DFP+94]. The paper ([Wie01]) describes application-specific tuning and several general techniques that allow the simulation to work well without excessive overhead for the optimistic synchronization. The Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) [Met] is another optimistic simulation engine that has been used for large-scale war-gaming exercises. The Scalable Simulation Framework (SSF) [CLL+99] is a simulation framework with pessimistic synchronization that has been used for large-scale modeling of the Internet.

The CSWEB application described in [YWC+96] simulates the voltage output of combinational digital circuits (that is, circuits without feedback paths). The circuit is partitioned into subcircuits; associated with each are input signal ports and output voltage ports, which are connected to form a representation of the whole circuit. The simulation of each subcircuit proceeds in a timestepped fashion; at each time step, the subcircuit's behavior depends on its previous state and the values read at its input ports (which correspond to values at the corresponding output ports of other subcircuits at previous time steps). Simulation of these subcircuits can proceed concurrently, with ordering constraints imposed by the relationship between values generated for output ports and values read on input ports. The solution described in [YWC+96] fits the *Event-Based Coordination* pattern, defining a task for each subcircuit and representing the ordering constraints as events.

Related Patterns

This pattern is similar to the *Pipeline* pattern in that both patterns apply to problems in which it is natural to decompose the computation into a collection of semi-independent entities interacting in terms of a flow of data. There are two key differences. First, in the *Pipeline* pattern, the interaction among entities is fairly regular, with all stages of the pipeline proceeding in a loosely synchronous way, whereas in the *Event-Based Coordination* pattern there is no such requirement, and the entities can interact in very irregular and asynchronous ways. Second, in the *Pipeline* pattern, the overall structure (number of tasks and their interaction) is usually fixed, whereas in the *Event-Based Coordination* pattern, the problem structure can be more dynamic.