CHAPTER 5

The *Supporting Structures* Design Space

5.1 INTRODUCTION

The *Finding Concurrency* and *Algorithm Structure* design spaces focus on algorithm expression. At some point, however, algorithms must be translated into programs. The patterns in the *Supporting Structures* design space address that phase of the parallel program design process, representing an intermediate stage between the problem-oriented patterns of the *Algorithm Structure* design space and the specific programming mechanisms described in the *Implementation Mechanisms* design space. We call these patterns *Supporting Structures* because they describe software constructions or "structures" that support the expression of parallel algorithms. An overview of this design space and its place in the pattern language is shown in Fig. 5.1.

The two groups of patterns in this space are those that represent program-structuring approaches and those that represent commonly used shared data structures. These patterns are briefly described in the next section. In some programming environments, some of these patterns are so well-supported that there is little work for the programmer. We nevertheless document them as patterns for two reasons: First, understanding the low-level details behind these structures is important for effectively using them. Second, describing these structures as patterns provides guidance for programmers who might need to implement them from scratch. The final section of this chapter describes structures that were not deemed important

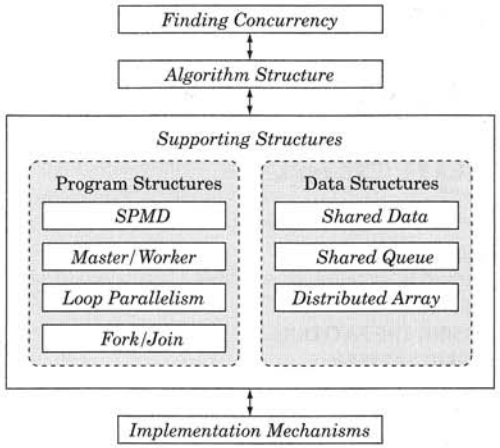

Figure 5.1: Overview of the *Supporting Structures* design space and its place in the pattern language

enough, for various reasons, to warrant a dedicated pattern, but which deserve mention for completeness.

5.1.1 Program Structuring Patterns

Patterns in this first group describe approaches for structuring source code. These patterns include the following.

- ***SPMD.*** In an *SPMD* (*Single Program, Multiple Data*) program, all UEs execute the same program (*Single Program*) in parallel, but each has its own set of data (*Multiple Data*). Different UEs can follow different paths through the program. In the source code, the logic to control this is expressed using a parameter that uniquely labels each UE (for example a process ID).

- ***Master/Worker.*** A master process or thread sets up a pool of worker processes or threads and a *bag of tasks*. The workers execute concurrently, with each worker repeatedly removing a task from the bag of tasks and processing it, until all tasks have been processed or some other termination condition has been reached. In some implementations, no explicit master is present.

- ***Loop Parallelism.*** This pattern addresses the problem of transforming a serial program whose runtime is dominated by a set of compute-intensive loops into a parallel program where the different iterations of the loop are executed in parallel.

- ***Fork/Join.*** A main UE forks off some number of other UEs that then continue in parallel to accomplish some portion of the overall work. Often the forking UE waits until the child UEs terminate and join.

While we define each of these program structures as a distinct pattern, this is somewhat artificial. It is possible, for example, to implement the *Master/Worker* pattern using the *Fork/Join* pattern or the *SPMD* pattern. These patterns do not represent exclusive, unique ways to structure a parallel program. Rather, they define the major idioms used by experienced parallel programmers.

These patterns also inevitably express a bias rooted in the subset of parallel programming environments we consider in this pattern language. To an MPI programmer, for example, all program structure patterns are essentially a variation on the *SPMD* pattern. To an OpenMP programmer, however, there is a huge difference between programs that utilize thread IDs (that is, the *SPMD* pattern) versus programs that express all concurrency in terms of loop-level worksharing constructs (that is, the *Loop Parallelism* pattern).

Therefore, in using these patterns, don't think of them too rigidly. These patterns express important techniques and are worthy of consideration in isolation, but do not hesitate to combine them in different ways to meet the needs of a particular problem. For example, in the *SPMD* pattern, we will discuss parallel algorithms based on parallelizing loops but expressed with the *SPMD* pattern. It might seem that this indicates that the *SPMD* and *Loop Parallelism* patterns are not really distinct patterns, but in fact it shows how flexible the *SPMD* pattern is.

5.1.2 Patterns Representing Data Structures

Patterns in this second group have to do with managing data dependencies. The *Shared Data* pattern deals with the general case. The others describe specific frequently used data structures.

- ***Shared Data.*** This pattern addresses the general problem of handling data that is shared by more than one UE, discussing both correctness and performance issues.

- ***Shared Queue.*** This pattern represents a "thread-safe" implementation of the familiar queue abstract data type (ADT), that is, an implementation of the queue ADT that maintains the correct semantics even when used by concurrently executing UEs.

- ***Distributed Array.*** This pattern represents a class of data structures often found in parallel scientific computing, namely arrays of one or more dimensions that are decomposed into subarrays and distributed among processes or threads.

5.2 FORCES

All of the program structuring patterns address the same basic problem: how to structure source code to best support algorithm structures of interest. Unique forces are applicable to each pattern, but in designing a program around these structures, there are some common forces to consider in most cases:

- **Clarity of abstraction.** Is the parallel algorithm clearly apparent from the source code?

In a well-structured program, the algorithm leaps from the page. The reader can see the details of the algorithm with little mental effort. We refer to this quality as *clarity of abstraction*. Good clarity of abstraction is always important for writing correct code, but is particularly essential for parallel programs: Parallel programmers must deal with multiple simultaneous tasks that interact in subtle ways. Getting this right can be very difficult, especially if it is hard to figure out what the algorithm is doing by looking at the source code.

- **Scalability.** How many processors can the parallel program effectively utilize?

  The scalability of a program is restricted by three factors. First, there is the amount of concurrency available in the algorithm. If an algorithm only has ten concurrent tasks, then running with more than ten PEs will provide no benefit. Second, the fraction of the runtime spent doing inherently serial work limits how many processors can be used. This is described quantitatively by Amdahl's law as discussed in Chapter 2. Finally, the parallel overhead of the algorithm contributes to the serial fraction mentioned in Amdahl's law and limits scalability.

- **Efficiency.** How close does the program come to fully utilizing the resources of the parallel computer? Recall the quantitative definition of efficiency given in Chapter 2:

$$E(P) = \frac{S(P)}{P} \tag{5.1}$$

$$= \frac{T(1)}{P\,T(P)} \tag{5.2}$$

  P is the number of PEs, $T(1)$ is some sequential reference time, and $T(P)$ is the parallel time with P PEs. $S(P)$ is the speedup.

  The most rigorous definition of efficiency sets $T(1)$ to the execution time of the best sequential algorithm corresponding to the parallel algorithm under study. When analyzing parallel programs, "best" sequential algorithms are not always available, and it is common to use the runtime for the parallel program on a single PE as the reference time. This can inflate the efficiency because managing the parallel computation always incurs some overhead, even when executing on a single PE. Efficiency is closely related to scalability because every highly scalable algorithm is also highly efficient. Even when the scalability is limited by the available number of tasks or the parallel hardware, however, algorithms can differ in their efficiency.

- **Maintainability.** Is the program easy to debug, verify, and modify?

  Casting an algorithm into source code is almost never a one-time proposition. Programs need to be debugged, new features added, performance tuned, etc. These changes to the source code are referred to as *maintenance*. Programs are more or less maintainable depending on how hard it is to make these changes and to do them correctly.

- **Environmental affinity.** Is the program well aligned with the programming environment and hardware of choice?

  If the hardware, for example, lacks support for shared memory, an algorithm structure based on shared memory would be a poor choice. This issue also comes up when considering programming environments. When creating a programming environment, the creators usually have a particular style of programming in mind. For example, OpenMP is designed specifically for programs consisting of a series of loops, the iterations of which will be split between multiple threads (loop-based parallelism). It is much easier to write software when the program structure employed is well aligned with the programming environment.

- **Sequential equivalence.** Where appropriate, does a program produce equivalent results when run with many UEs as with one? If not equivalent, is the relationship between them clear?

  It is highly desirable that the results of an execution of a parallel program be the same regardless of the number of PEs used. This is not always possible, especially if bitwise equivalence is desired, because floating-point operations performed in a different order can produce small (or, for ill-conditioned algorithms, not so small) changes in the resulting values. However, if we know that the parallel program gives equivalent results when executed on one processor as many, then we can reason about correctness and do most of the testing on the single-processor version. This is much easier, and thus, when possible, sequential equivalence is a highly desirable goal.

5.3 CHOOSING THE PATTERNS

Choosing which program structure pattern to use is usually straightforward. In most cases, the programming environment selected for the project and the patterns used from the *Algorithm Structure* design space point to the appropriate program structure pattern to use. We will consider these two factors separately.

The relationship between the patterns in the *Algorithm Structure* and *Supporting Structures* design spaces is shown in Table 5.1. Notice that the *Supporting*

Table 5.1: Relationship between *Supporting Structures* patterns and *Algorithm Structure* patterns. The number of stars (ranging from zero to four) is an indication of the likelihood that the given *Supporting Structures* pattern is useful in the implementation of the *Algorithm Structure* pattern.

	Task Parallelism	*Divide and Conquer*	*Geometric Decomposition*	*Recursive Data*	*Pipeline*	*Event-Based Coordination*
SPMD	★★★★	★★★	★★★★	★★	★★★	★★
Loop Parallelism	★★★★	★★	★★★			
Master/Worker	★★★★	★★	★	★	★	★
Fork/Join	★★	★★★★	★★		★★★★	★★★★

Table 5.2: Relationship between *Supporting Structures* patterns and programming environments. The number of stars (ranging from zero to four) is an indication of the likelihood that the given *Supporting Structures* pattern is useful in the programming environment.

	OpenMP	MPI	Java
SPMD	★★★	★★★★	★★
LoopParallelism	★★★★	★	★★★
Master/Worker	★★	★★★	★★★
Fork/Join	★★★		★★★★

Structures patterns can be used with multiple *Algorithm Structure* patterns. For example, consider the range of applications using the *Master/Worker* pattern: In [BCM+91, CG91, CGMS94], it is used to implement everything from embarrassingly parallel programs (a special case of the *Task Parallelism* pattern) to those using the *Geometric Decomposition* pattern. The *SPMD* pattern is even more flexible and covers the most important algorithm structures used in scientific computing (which tends to emphasize the *Geometric Decomposition, Task Parallelism,* and *Divide and Conquer* patterns). This flexibility can make it difficult to choose a program structure pattern solely on the basis of the choice of *Algorithm Structure* pattern(s).

The choice of programming environment, however, helps narrow the choice considerably. In Table 5.2, we show the relationship between programming environments and the *Supporting Structures* patterns. MPI, the programming environment of choice on any distributed-memory computer, strongly favors the *SPMD* pattern. OpenMP, the standard programming model used on virtually every shared-memory computer on the market, is closely aligned with the *Loop Parallelism* pattern. The combination of programming environment and *Algorithm Structure* patterns typically selects which *Supporting Structures* patterns to use.

5.4 THE *SPMD* PATTERN

Problem

The interactions between the various UEs cause most of the problems when writing correct and efficient parallel programs. How can programmers structure their parallel programs to make these interactions more manageable and easier to integrate with the core computations?

Context

A parallel program takes complexity to a new level. There are all the normal challenges of writing any program. On top of those challenges, the programmer must manage multiple tasks running on multiple UEs. In addition, these tasks and UEs interact, either through exchange of messages or by sharing memory. In spite of these complexities, the program must be correct, and the interactions must be well orchestrated if excess overhead is to be avoided.

Fortunately, for most parallel algorithms, the operations carried out on each UE are similar. The data might be different between UEs, or slightly different computations might be needed on a subset of UEs (for example, handling boundary conditions in partial differential equation solvers), but for the most part each UE will carry out similar computations. Hence, in many cases the tasks and their interactions can be made more manageable by bringing them all together into one source tree. This way, the logic for the tasks is side by side with the logic for the interactions between tasks, thereby making it much easier to get them right.

This is the so-called "Single Program, Multiple Data" (SPMD) approach. It emerged as the dominant way to structure parallel programs early in the evolution of scalable computing, and programming environments, notably MPI, have been designed to support this approach.[1]

In addition to the advantages to the programmer, SPMD makes management of the solution much easier. It is much easier to keep a software infrastructure up to date and consistent if there is only one program to manage. This factor becomes especially important on systems with large numbers of PEs. These can grow to huge numbers. For example, the two fastest computers in the world according to the November 2003 top 500 list [Top], the Earth Simulator at the Earth Simulator Center in Japan and the ASCI Q at Los Alamos National Labs, have 5120 and 8192 processors, respectively. If each PE runs a distinct program, managing the application software could quickly become prohibitively difficult.

> *This pattern is by far the most commonly used pattern for structuring parallel programs. It is particularly relevant for MPI programmers and problems using the* Task Parallelism *and* Geometric Decomposition *patterns. It has also proved effective for problems using the* Divide and Conquer *and* Recursive Data *patterns.*

Forces

- Using similar code for each UE is easier for the programmer, but most complex applications require that different operations run on different UEs and with different data.
- Software typically outlives any given parallel computer. Hence, programs should be portable. This compels the programmer to assume the lowest common denominator in programming environments, and to assume that only basic mechanisms for coordinating tasks will be available.

[1] It is not that the available programming environments pushed SPMD; the force was the other way around. The programming environments for MIMD machines pushed SPMD because that is the way programmers wanted to write their programs. They wrote them this way because they found it to be the best way to get the logic correct and efficient for what the tasks do and how they interact. For example, the programming environment PVM, sometimes considered a predecessor to MPI, in addition to the SPMD program structure also supported running different programs on different UEs (sometimes called the MPMD program structure). The MPI designers, with the benefit of the PVM experience, chose to support only SPMD.

- Achieving high scalability and good efficiency in a parallel program requires that the program be well aligned with the architecture of the parallel computer. Therefore, the details of the parallel system must be exposed and, where appropriate, under the programmer's control.

Solution

The *SPMD* pattern solves this problem by creating a single source-code image that runs on each of the UEs. The solution consists of the following basic elements.

- **Initialize.** The program is loaded onto each UE and opens with bookkeeping operations to establish a common context. The details of this procedure are tied to the parallel programming environment and typically involve establishing communication channels with other UEs.
- **Obtain a unique identifier.** Near the top of the program, an identifier is set that is unique to each UE. This is usually the UE's rank within the MPI group (that is, a number in the interval from 0 to $N - 1$, where N is the number of UEs) or the thread ID in OpenMP. This unique identifier allows different UEs to make different decisions during program execution.
- **Run the same program on each UE, using the unique ID to differentiate behavior on different UEs.** The same program runs on each UE. Differences in the instructions executed by different UEs are usually driven by the identifier. (They could also depend on the UE's data.) There are many ways to specify that different UEs take different paths through the source code. The most common are (1) branching statements to give specific blocks of code to different UEs and (2) using the UE identifier in loop index calculations to split loop iterations among the UEs.
- **Distribute data.** The data operated on by each UE is specialized to that UE by one of two techniques: (1) decomposing global data into chunks and storing them in the local memory of each UE, and later, if required, recombining them into the globally relevant results or (2) sharing or replicating the program's major data structures and using the UE identifier to associate subsets of the data with particular UEs.
- **Finalize.** The program closes by cleaning up the shared context and shutting down the computation. If globally relevant data was distributed among UEs, it will need to be recombined.

Discussion. An important issue to keep in mind when developing SPMD programs is the *clarity of abstraction*, that is, how easy it is to understand the algorithm from reading the program's source code. Depending on how the data is handled, this can range from awful to good. If complex index algebra on the UE identifier is needed to determine the data relevant to a UE or the instruction branch, the algorithm can be almost impossible to follow from the source code. (The *Distributed Array* pattern discusses useful techniques for arrays.)

In some cases, a replicated data algorithm combined with simple loop splitting is the best option because it leads to a clear abstraction of the parallel algorithm within the source code and an algorithm with a high degree of sequential equivalence. Unfortunately, this simple approach might not scale well, and more complex solutions might be needed. Indeed, SPMD algorithms can be highly scalable, and algorithms requiring complex coordination between UEs and scaling out to several thousand UEs [PH95] have been written using this pattern. These highly scalable algorithms are usually extremely complicated as they distribute the data across the nodes (that is, no simplifying replicated data techniques), and they generally include complex load-balancing logic. These algorithms, unfortunately, bear little resemblance to their serial counterparts, reflecting a common criticism of the *SPMD* pattern.

An important advantage of the *SPMD* pattern is that overheads associated with startup and termination are segregated at the beginning and end of the program, not inside time-critical loops. This contributes to efficient programs and results in the efficiency issues being driven by the communication overhead, the capability to balance the computational load among the UEs, and the amount of concurrency available in the algorithm itself.

SPMD programs are closely aligned with programming environments based on message passing. For example, most MPI or PVM programs use the *SPMD* pattern. Note, however, that it is possible to use the *SPMD* pattern with OpenMP [CPP01]. With regard to the hardware, the *SPMD* pattern does not assume anything concerning the address space within which the tasks execute. As long as each UE can run its own instruction stream operating on its own data (that is, the computer can be classified as MIMD), the SPMD structure is satisfied. This generality of SPMD programs is one of the strengths of this pattern.

Examples

The issues raised by application of the *SPMD* pattern are best discussed using three specific examples:

- Numerical integration to estimate the value of a definite integral using the trapezoid rule
- Molecular dynamics, force computations
- Mandelbrot set computation

Numerical integration. We can use a very simple program, frequently used in teaching parallel programming, to explore many of the issues raised by the *SPMD* pattern. Consider the problem of estimating the value of π using Eq. 5.3.

$$\pi = \int_0^1 \frac{4}{1 + x^2}\, dx \tag{5.3}$$

We use trapezoidal integration to numerically solve the integral. The idea is to fill the area under a curve with a series of rectangles. As the width of the rectangles

```
#include <stdio.h>
#include <math.h>

int main () {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi %lf\n",pi);
   return 0;
}
```

Figure 5.2: Sequential program to carry out a trapezoid rule integration to compute $\int_0^1 \frac{4}{1+x^2}\,dx$

approaches zero, the sum of the areas of the rectangles approaches the value of the integral.

A program to carry this calculation out on a single processor is shown in Fig. 5.2. To keep the program as simple as possible, we fix the number of steps to use in the integration at 1,000,000. The variable `sum` is initialized to 0 and the step size is computed as the range in `x` (equal to 1.0 in this case) divided by the number of steps. The area of each rectangle is the width (the step size) times the height (the value of the integrand at the center of the interval). Because the width is a constant, we pull it out of the summation and multiply the sum of the rectangle heights by the step size, `step`, to get our estimate of the definite integral.

We will look at several versions of the parallel algorithm. We can see all the elements of a classic SPMD program in the simple MPI version of this program, as shown in Fig. 5.3. The same program is run on each UE. Near the beginning of the program, the MPI environment is initialized and the ID for each UE (`my_id`) is given by the process rank for each UE in the process group associated with the communicator `MPI_COMM_WORLD` (for information about communicators and other MPI details, see the MPI appendix, Appendix B). We use the number of UEs and the ID to assign loop ranges (`i_start` and `i_end`) to each UE. Because the number of steps may not be evenly divided by the number of UEs, we have to make sure the last UE runs up to the last step in the calculation. After the partial sums have been computed on each UE, we multiply by the step size, `step`, and then use the `MPI_Reduce()` routine to combine the partial sums into a global sum. (Reduction operations are described in more detail in the *Implementation Mechanisms* design space.) This global value will only be available in the process with `my_id == 0`, so we direct that process to print the answer.

In essence, what we have done in the example in Fig. 5.3 is to replicate the key data (in this case, the partial summation value, `sum`), use the UE's ID to explicitly


```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
   int i, i_start, i_end;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   int my_id, numprocs;
   step = 1.0/(double) num_steps;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

   i_start = my_id * (num_steps/numprocs);
   i_end = i_start + (num_steps/numprocs);
   if (my_id == (numprocs-1)) i_end = num_steps;

   for (i=i_start; i< i_end; i++)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   sum *= step;
   MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
   if (my_id == 0) printf("pi %lf\n",pi);
   MPI_Finalize();
   return 0;
}
```

Figure 5.3: MPI program to carry out a trapezoid rule integration in parallel by assigning one block of loop iterations to each UE and performing a reduction

split up the work into blocks with one block per UE, and then recombine the local results into the final global result. The challenge in applying this pattern is to (1) split up the data correctly, (2) correctly recombine the results, and (3) achieve an even distribution of the work. The first two steps were trivial in this example. The load balance, however, is a bit more difficult. Unfortunately, the simple procedure we used in Fig. 5.3 could result in significantly more work for the last UE if the number of UEs does not evenly divide the number of steps. For a more even distribution of the work, we need to spread out the extra iterations among multiple UEs. We show one way to do this in the program fragment in Fig. 5.4. We compute the number of iterations left over after dividing the number of steps by the number of processors (`rem`). We will increase the number of iterations computed by the first `rem` UEs to cover that amount of work. The code in Fig. 5.4 accomplishes that task. These sorts of index adjustments are the bane of programmers using the *SPMD* pattern. Such code is error-prone and the source of hours of frustration as program readers try to understand the reasoning behind this logic.

Finally, we use a loop-splitting strategy for the numerical integration program. The resulting program is shown in Fig. 5.5. This approach uses a common trick to

```
int rem = num_steps % numprocs;

i_start = my_id * (num_steps/numprocs);
i_end = i_start + (num_steps/numprocs);

if (rem != 0){
   if(my_id < rem){
      i_start += my_id;
      i_end += (my_id + 1);
   }
   else {
      i_start += rem;
      i_end += rem;
   }
}
```

Figure 5.4: Index calculation that more evenly distributes the work when the number of steps is not evenly divided by the number of UEs. The idea is to split up the remaining tasks (`rem`) among the first `rem` UEs.

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   int my_id, numprocs;
   step = 1.0/(double) num_steps;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

   for (i=my_id; i< num_steps; i+= numprocs)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   sum *= step;
   MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
   if (my_id == 0) printf("pi %lf\n",pi);
   MPI_Finalize();
   return 0;
}
```

Figure 5.5: MPI program to carry out a trapezoid rule integration in parallel using a simple loop-splitting algorithm with cyclic distribution of iterations and a reduction


```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main () {
   int num_steps = 1000000;
   double pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

#pragma omp parallel reduction(+:sum)
   {
      int i, id = omp_get_thread_num();
      int numthreads = omp_get_num_threads();
      double x;

      for (i=id;i< num_steps; i+=numthreads){
         x = (i+0.5)*step;
         sum += + 4.0/(1.0+x*x);
      }
   } // end of parallel region
   pi = step * sum;
   printf("\n pi is %lf\n",pi);
   return 0;
}
```

Figure 5.6: OpenMP program to carry out a trapezoid rule integration in parallel using the same SPMD algorithm used in Fig. 5.5

achieve a cyclic distribution of the loop iterations: Each UE starts with the iteration equal to its rank, and then marches through the iterations of the loop with a stride equal to the number of UEs. The iterations are interleaved among the UEs, in the same manner as a deck of cards would be dealt. This version of the program evenly distributes the load without resorting to complex index algebra.

SPMD programs can also be written using OpenMP and Java. In Fig. 5.6, we show an OpenMP version of our trapezoidal integration program. This program is very similar to the analogous MPI program. The program has a single parallel region. We start by finding the thread ID and the number of threads in the team. We then use the same trick to interleave iterations among the team of threads. As with the MPI program, we use a reduction to combine partial sums into a single global sum.

Molecular dynamics. Throughout this pattern language, we have used molecular dynamics as a recurring example. Molecular dynamics simulates the motions of a large molecular system. It uses an explicit time-stepping methodology where at each time step, the force on each atom is computed and standard techniques from classical mechanics are used to compute how the forces change atomic motions.

This problem is ideal for presenting key concepts in parallel algorithms because there are so many ways to approach the problem based on the target computer system and the intended use of the program. In this discussion, we will follow the approach taken in [Mat95] and assume that (1) a sequential version of the program

```
Int const N // number of atoms

Array of Real :: atoms (3,N) //3D coordinates
Array of Real :: velocities (3,N) //velocity vector
Array of Real :: forces (3,N) //force in each dimension
Array of List :: neighbors(N) //atoms in cutoff volume

loop over time steps
   initialize_forces (N, Forces)
   if(time to update neighbor list)
       neighbor_list (N, Atoms, neighbors)
   end if
   vibrational_forces (N, atoms, forces)
   rotational_forces (N, atoms, forces)
   non_bonded_forces (N, atoms, neighbors, forces)
   update_atom_positions_and_velocities(
                  N, atoms, velocities, forces)
   physical_properties ( ... Lots of stuff ... )
end loop
```

Figure 5.7: Pseudocode for molecular dynamics example. This code is very similar to the version discussed earlier, but a few extra details have been included. To support more detailed pseudocode examples, the call to the function that initializes the force arrays has been made explicit. Also, the fact that the neighbor list is only occasionally updated is made explicit.

exists, (2) having a single program for sequential and parallel execution is important, and (3) the target system is a small cluster connected by standard Ethernet LAN. More scalable algorithms for execution on massively parallel systems are discussed in [PH95].

The core algorithm, including pseudocode, was presented in Sec. 3.1.3. While we won't repeat the discussion here, we do provide a copy of the pseudocode in Fig. 5.7.

The parallel algorithm is discussed in several of the patterns in the *Finding Concurrency* and *Algorithm Structure* design spaces. Following are the key points from those discussions that we will need here along with the location of the original discussion.

1. Computing the `non_bonded_forces` takes the overwhelming majority of the runtime (Sec. 3.1.3).
2. In computing the `non_bonded_force`, each atom potentially interacts with all the other atoms. Hence, each UE needs read access to the full atomic position array. Also, due to Newton's third law, each UE will be scattering contributions to the force across the full force array (the Examples section of the *Data Sharing* pattern).
3. One way to decompose the MD problem into tasks is to focus on the computations needed for a particular atom, that is, we can parallelize this problem by assigning atoms to UEs (the Examples section of the *Task Decomposition* pattern).

Given that our target is a small cluster and from point (1) in the preceding list, we will only parallelize the force computations. Because the network is slow for

parallel computing and given the data dependency in point (2), we will:

- Keep a copy of the full force and coordinate arrays on each node.
- Have each UE redundantly update positions and velocities for the atoms (that is, we assume it is cheaper to redundantly compute these terms than to do them in parallel and communicate the results).
- Have each UE compute its contributions to the force array and then combine (or reduce) the UEs' contributions into a single global force array copied onto each UE.

The algorithm is a simple transformation from the sequential algorithm. Pseudocode for this SPMD program is shown in Fig. 5.8. As with any MPI program,

```
#include <mpi.h>

    Int const N // number of atoms
    Int const LN // maximum number of atoms assigned to a UE

    Int ID // an ID for each UE
    Int num_UEs // the number of UEs in the parallel computation

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: velocities (3,N) //velocity vector
    Array of Real :: forces (3,N) //force in each dimension
    Array of Real :: final_forces(3,N) //globally summed force
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms for this UE

    ID = 0 // default ID (used by the serial code)
    num_UEs = 1 // default num_UEs (used by the serial code)
    MPI_Init()
    MPI_Comm_size(MPI_COMM_WORLD, &ID)
    MPI_Comm_rank(MPI_COMM_WORLD, &num_UEs)

    loop over time steps
       initialize_forces (N, forces, final_forces)
       if(time to update neighbor list)
          neighbor_list (N, LN, atoms, neighbors)
       end if
       vibrational_forces (N, LN, local_atoms, atoms, forces)
       rotational_forces (N, LN, local_atoms, atoms, forces)
       non_bonded_forces (N, LN, atoms, local_atoms, neighbors,
          forces)

       MPI_All_reduce{forces, final_forces, 3*N, MPI_REAL,
          MPI_SUM, MPI_COMM_WORLD)

       update_atom_positions_and_velocities(
                      N, atoms, velocities, final_forces)
       physical_properties ( ... Lots of stuff ... )
    end loop

    MPI_Finalize()
```

Figure 5.8: Pseudocode for an SPMD molecular dynamics program using MPI

```
function non_bonded_forces (N, LN, atoms, local_atoms,
                                    neighbors, Forces)

    Int N // number of atoms
    Int LN // maximum number of atoms assigned to a UE

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: forces (3,N) //force in each dimension
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms assigned to this UE
    real :: forceX, forceY, forceZ

    loop [i] over local_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 5.9: Pseudocode for the nonbonded computation in a typical parallel molecular dynamics code. This code is almost identical to the sequential version of the function shown in Fig. 4.4. The only major change is a new array of integers holding the indices for the atoms assigned to this UE, `local_atoms`. We've also assumed that the neighbor list has been generated to hold only those atoms assigned to this UE. For the sake of allocating space for these arrays, we have added a parameter `LN` which is the largest number of atoms that can be assigned to a single UE.

the MPI include file is referenced at the top of the program. The MPI environment is initialized and the ID is associated with the rank of the MPI process.

Only a few changes are made to the sequential functions. First, a second force array called `final_forces` is defined to hold the globally consistent force array appropriate for the update of the atomic positions and velocities. Second, a list of atoms assigned to the UE is created and passed to any function that will be parallelized. Finally, the `neighbor_list` is modified to hold the list for only those atoms assigned to the UE.

Finally, within each of the functions to be parallelized (the forces calculations), the loop over atoms is replaced by a loop over the list of local atoms.

We show an example of these simple changes in Fig. 5.9. This is almost identical to the sequential version of this function discussed in the *Task Parallelism* pattern. As discussed earlier, the following are the key changes.

- A new array has been added to hold indices for the atoms assigned to this UE. This array is of length `LN` where `LN` is the maximum number of atoms that can be assigned to a single UE.

- The loop over all atoms (loop over `i`) has been replaced by a loop over the elements of the `local_atoms` list.
- We assume that the neighbor list has been modified to correspond to the atoms listed in the `local_atoms` list.

The resulting code can be used for a sequential version of the program by setting `LN` to `N` and by putting the full set of atom indices into `local_atoms`. This feature satisfies one of our design goals: that a single source code would work for both sequential and parallel versions of the program.

The key to this algorithm is in the function to compute the neighbor list. The neighbor list function contains a loop over the atoms. For each atom `i`, there is a loop over all other atoms and a test to determine which atoms are in the neighborhood of atom `i`. The indices for these neighboring atoms are saved in `neighbors`, a list of lists. Pseudocode for this code is shown in Fig. 5.10.

```
function neighbor (N, LN, ID, cutoff, atoms, local_atoms,
                   neighbors)

    Int N // number of atoms
    Int LN // max number of atoms assigned to a UE
    Real cutoff // radius of sphere defining neighborhood

    Array of Real :: atoms (3,N) //3D coordinates
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms assigned to this UE
    real :: dist_squ

    initialize_lists (local_atoms, neighbors)

    loop [i] over atoms on UE //split loop iterations among UEs
        add_to_list (i, local_atoms)
        loop [j] over atoms greater than i
            dist_squ = square(atom(1,i)-atom(1,j)) +
                       square(atom(2,i)-atom(2,j)) +
                       square(atom(3,i)-atom(3,j))
            if(dist_squ < (cutoff * cutoff))
                add_to_list (j, neighbors(i))
            end if
        end loop [j]

    end loop [i]
end function neighbors
```

Figure 5.10: Pseudocode for the neighbor list computation. For each atom `i`, the indices for atoms within a sphere of radius `cutoff` are added to the neighbor list for atom `i`. Notice that the second loop (over `j`) only considers atoms with indices greater than `i`. This accounts for the symmetry in the force computation due to Newton's third law of motion, that is, that the force between atom `i` and atom `j` is just the negative of the force between atom `j` and atom `i`.

The logic defining how the parallelism is distributed among the UEs is captured in the single loop in Fig. 5.10:

```
loop [i] over atoms on UE //split loop iterations among UEs
   add_to_list (i, local_atoms)
```

The details of how this loop is split among UEs depends on the programming environment. An approach that works well with MPI is the cyclic distribution we used in Fig. 5.5:

```
for (i=id;i<number_of_atoms; i+= number_of_UEs){
   add_to_list (i, local_atoms)
}
```

More complex or even dynamic distributions can be handled by creating an *owner-computes filter* [Mat95]. An owner-computes filter provides a flexible and reusable schedule for mapping loop iterations onto UEs. The filter is a boolean function of the ID and the loop iteration. The value of the function depends on whether a UE "owns" a particular iteration of a loop. For example, in a molecular dynamics program, the call to the owner-computes function would be added at the top of the parallelized loops over atoms:

```
for (i=0;i<number_of_atoms; i++){
   if !(is_owner (i)) break

   add_to_list (i, local_atoms)
}
```

No other changes to the loop are needed to support expression of concurrency. If the logic managing the loop is convoluted, this approach partitions the iterations among the UEs without altering that logic, and the index partitioning logic is located clearly in one place in the source code. Another advantage occurs when several loops that should be scheduled the same way are spread out throughout a program. For example, on a NUMA machine or a cluster it is very important that data brought close to a PE be used as many times as possible. Often, this means reusing the same schedule in many loops.

This approach is described further for molecular dynamics applications in [Mat95]. It could be important in this application since the workload captured in the neighbor list generation may not accurately reflect the workload in the various force computations. One could easily collect information about the time required for each atom and then readjust the `is_owner` function to produce more optimal work loads.


```
#include <omp.h>
   Int const N // number of atoms
   Int const LN // maximum number of atoms assigned to a UE
   Int ID // an ID for each UE
   Int num_UEs // number of UEs in the parallel computation
   Array of Real :: atoms(3,N) //3D coordinates
   Array of Real :: velocities(3,N) //velocity vector
   Array of Real :: forces(3,N) //force in each dim
   Array of List :: neighbors(LN) //atoms in cutoff volume
   Array of Int :: local_atoms(LN) //atoms for this UE

   ID = 0
   num_UEs = 1

#pragma omp parallel private (ID, num_UEs, local_atoms, forces) {
   ID = omp_get_thread_num()
   num_UEs = omp_get_num_threads()

   loop over time steps
      initialize_forces (N, forces, final_forces)
      if(time to update neighbor list)
         neighbor_list (N, LN, atoms, neighbors)
      end if
      vibrational_forces (N, LN, local_atoms, atoms, forces)
      rotational_forces (N, LN, local_atoms, atoms, forces)
      non_bonded_forces (N, LN, atoms, local_atoms,
                                        neighbors, forces)
#pragma critical
      final_forces += forces
#barrier

#pragma single
{
      update_atom_positions_and_velocities(
                        N, atoms, velocities, forces)
      physical_properties ( ... Lots of stuff ... )
} // remember, the end of a single implies a barrier

   end loop

} // end of OpenMP parallel region
```

Figure 5.11: Pseudocode for a parallel molecular dynamics program using OpenMP

These SPMD algorithms work for OpenMP programs as well. All of the basic functions remain the same. The top-level program is changed to reflect the needs of OpenMP. This is shown in Fig. 5.11.

The loop over time is placed inside a single parallel region. The parallel region is created with the `parallel` pragma:

```
#pragma omp parallel private (ID, num_UEs, local_atoms, forces)
```

This pragma causes a team of threads to be created with each member of the team executing the loop over time. The `private` clause causes copies of the listed variables to be created for each UE. The reduction is carried out in a critical section:

```
#pragma critical
        final_forces += forces
```

A `reduction` clause on the parallel region cannot be used in this case because the result would not be available until the parallel region completes. The critical section produces the correct result, but the algorithm used has a runtime that is linear in the number of UEs and is hence suboptimal relative to other reduction algorithms as discussed in the *Implementation Mechanisms* design space. On systems with a modest number of processors, however, the reduction with a critical section works adequately.

The barrier following the critical section is required to make sure the reduction completes before the atomic positions and velocities are updated. We then use an OpenMP `single` construct to cause only one UE to do the update. An additional barrier is not needed following the `single` since the close of a `single` construct implies a barrier. The functions used to compute the forces are unchanged between the OpenMP and MPI versions of the program.

Mandelbrot set computation. Consider the well-known Mandelbrot set [Dou86]. We discussed this problem and its parallelization as a task-parallel problem in the *Task Parallelism* pattern. Each pixel is colored based on the behavior of the quadratic recurrence relation in Eq. 5.4.

$$Z_{n+1} = Z_n^2 + C \tag{5.4}$$

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 (-1.5 to 1.5) and the real part on the horizontal axis (-1 to 2). 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.

In the *Task Parallelism* pattern, we described a parallel algorithm where each task corresponds to the computation of a row in the image. A static schedule with more tasks than UEs should be possible that achieves an effective statistical balance of the load among nodes. We will show how to solve this problem using the *SPMD* pattern with MPI.

Pseudocode for the sequential version of this code is shown in Fig. 5.12. The interesting part of the problem is hidden inside the routine `compute_Row()`. Because the details of this routine are not important for understanding the parallel algorithm, we will not show them here, however. At a high level, for each point in the row the following happens.


```
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map

Real :: conv // divergence rate for a pixel
Array of Int :: color_map (M) // pixel color based on conv rate
Array of Int :: row (RowSize) // Pixels to draw
Array of Real :: ranges(2) // ranges in X and Y dimensions

manage_user_input(ranges, color_map) // input ranges, color map
initialize_graphics(RowSize, Nrows, M, ranges, color_map)

for (int i = 0; i<Nrows; i++){

   compute_Row (RowSize, ranges, row)

   graph(i, RowSize, M, color_map, ranges, row)

} // end loop [i] over rows
```

Figure 5.12: Pseudocode for a sequential version of the Mandelbrot set generation program

- Each pixel corresponds to a value of C in the quadratic recurrence. We compute this value based on the input `range` and the pixel indices.
- We then compute the terms in the recurrence and set the value of the pixel based on whether it converges to a fixed value or diverges. If it diverges, we set the pixel value based on the rate of divergence.

Once computed, the rows are plotted to make the well-known Mandelbrot set images. The colors used for the pixels are determined by mapping divergence rates onto a color map.

An SPMD program based on this algorithm is straightforward; code is shown in Fig. 5.13. We will assume the computation is being carried out on some sort of distributed-memory machine (a cluster or even an MPP) and that there is one machine that serves as the interactive graphics node, while the others are restricted to computation. We will assume that the graphics node is the one with rank 0.

The program starts with the usual MPI setup, as described in the MPI appendix, Appendix B. The UE with rank 0 takes input from the user and then broadcasts this to the other UEs. It then loops over the number of rows in the image, receiving rows as they finish and plotting them. UEs with rank other than 0 use a cyclic distribution of loop iterations and send the rows to the graphics UE as they finish.

Known uses. The overwhelming majority of MPI programs use this pattern. Pedagogically oriented discussions of SPMD programs and examples can be found in MPI textbooks such as [GLS99] and [Pac96]. Representative applications using this pattern include quantum chemistry [WSG95], finite element methods [ABKP03, KLK+03], and 3D gas dynamics [MHC+99].

```
#include <mpi.h>
    Int const Nrows // number of rows in the image
    Int const RowSize // number of pixels in a row
    Int const M // number of colors in color map
    Real :: conv // divergence rate for a pixel
    Array of Int :: color_map (M) // pixel color based on conv rate
    Array of Real :: ranges(2) // ranges in X and Y dimensions
    Int :: inRowSize // size of received row
    Int :: ID // ID of each UE (process)
    Int :: num_UEs // number of UEs (processes)
    Int :: nworkers // number of UEs computing rows
    MPI_Status :: stat // MPI status parameter

    MPI_Init()
    MPI_Comm_size(MPI_COMM_WORLD, &ID)
    MPI_Comm_rank(MPI_COMM_WORLD, &num_UEs)

// Algorithm requires at least two UEs since we are
// going to dedicate one to graphics
    if (num_UEs < 2) MPI_Abort(MPI_COMM_WORLD, 1)

    if (ID == 0 ){
       manage_user_input(ranges, color_map) // input ranges, color map
       initialize_graphics(RowSize, Nrows, M, ranges, color_map)
  }

 // Broadcast data from rank 0 process to all other processes
    MPI_Bcast (ranges, 2, MPI_REAL, 0, MPI_COMM_WORLD);
    if (ID == 0) { // UE with rank 0 does graphics
      for (int i = 0; i<Nrows; i++){
        MPI_Recv(row, &inRowSize, MPI_REAL, MPI_ANY_SOURCE,
                        MPI_ANY_TAG, MPI_COMM_WORLD, &stat)
        row_index = stat(MPI_TAG)
        graph(row_index, RowSize, M, color_map, ranges, Row)
      } // end loop over i
    else { // The other UEs compute the rows
      nworkers = num_UEs - 1
      for (int i = ID-1; i<Nrows; i+=nworkers){
        compute_Row (RowSize, ranges, row)
        MPI_Send (row, RowSize, MPI_REAL, 0, i, MPI_COMM_WORLD);
      } // end loop over i
    }
    MPI_Finalize()
```

Figure 5.13: Pseudocode for a parallel MPI version of the Mandelbrot set generation program

Examples of the *SPMD* pattern in combination with the *Distributed Array* pattern include the GAMESS quantum chemistry program [OSG03] and the ScaLAPACK library [BCC+97, Sca].

Related Patterns

The *SPMD* pattern is very general and can be used to implement other patterns. Many of the examples in the text of this pattern are closely related to the *Loop Parallelism* pattern. Most applications of the *Geometric Decomposition* pattern with MPI use the *SPMD* pattern as well. The *Distributed Array* pattern is essentially a special case of distributing data for programs using the *SPMD* pattern.

5.5 THE *MASTER/WORKER* PATTERN

Problem

How should a program be organized when the design is dominated by the need to dynamically balance the work on a set of tasks among the UEs?

Context

Parallel efficiency follows from an algorithm's parallel overhead, its serial fraction, and the load balancing. A good parallel algorithm must deal with each of these, but sometimes balancing the load is so difficult that it dominates the design. Problems falling into this category usually share one or more of the following characteristics.

- The workloads associated with the tasks are highly variable and unpredictable. If workloads are predictable, they can be sorted into equal-cost bins, statically assigned to UEs, and parallelized using the *SPMD* or *Loop Parallelism* patterns. But if they are unpredictable, static distributions tend to produce suboptimal load balance.

- The program structure for the computationally intensive portions of the problem doesn't map onto simple loops. If the algorithm is loop-based, one can usually achieve a statistically near-optimal workload by a cyclic distribution of iterations or by using a dynamic schedule on the loop (for example, in OpenMP, by using the `schedule(dynamic)` clause). But if the control structure in the program is more complex than a simple loop, more general approaches are required.

- The capabilities of the PEs available for the parallel computation vary across the parallel system, change over the course of the computation, or are unpredictable.

In some cases, tasks are tightly coupled (that is, they communicate or share read-and-write data) and must be active at the same time. In this case, the *Master/Worker* pattern is not applicable: The programmer has no choice but to *explicitly* size or group tasks onto UEs dynamically (that is, during the computation) to achieve an effective load balance. The logic to accomplish this can be difficult to implement, and if one is not careful, can add prohibitively large parallel overhead.

If the tasks are independent of each other, however, or if the dependencies can somehow be pulled out from the concurrent computation, the programmer has much greater flexibility in how to balance the load. This allows the load balancing to be done automatically and is the situation we address in this pattern.

> *This pattern is particularly relevant for problems using the* Task Parallelism *pattern when there are no dependencies among the tasks (embarrassingly parallel problems). It can also be used with the* Fork/Join *pattern for the cases where the mapping of tasks onto UEs is indirect.*

Forces

- The work for each task, and in some cases even the capabilities of the PEs, varies unpredictably in these problems. Hence, explicit predictions of the runtime for any given task are not possible and the design must balance the load without them.
- Operations to balance the load impose communication overhead and can be very expensive. This suggests that scheduling should revolve around a smaller number of large tasks. However, large tasks reduce the number of ways tasks can be partitioned among the PEs, thereby making it more difficult to achieve good load balance.
- Logic to produce an optimal load can be convoluted and require error-prone changes to a program. Programmers need to make trade-offs between the desire for an optimal distribution of the load and code that is easy to maintain.

Solution

The well-known *Master/Worker* pattern is a good solution to this problem. This pattern is summarized in Fig. 5.14. The solution consists of two logical elements: a *master* and one or more instances of a *worker*. The master initiates the computation and sets up the problem. It then creates a *bag of tasks*. In the classic algorithm, the master then waits until the job is done, consumes the results, and then shuts down the computation.

A straightforward approach to implementing the bag of tasks is with a single shared queue as described in the *Shared Queue* pattern. Many other mechanisms for creating a globally accessible structure where tasks can be inserted and removed are possible, however. Examples include a tuple space [CG91, FHA99], a distributed queue, or a monotonic counter (when the tasks can be specified with a set of contiguous integers).

Meanwhile, each worker enters a loop. At the top of the loop, the worker takes a task from the bag of tasks, does the indicated work, tests for completion, and then goes to fetch the next task. This continues until the termination condition is met, at which time the master wakes up, collects the results, and finishes the computation.

Master/worker algorithms automatically balance the load. By this, we mean the programmer does not explicitly decide which task is assigned to which UE. This decision is made dynamically by the master as a worker completes one task and accesses the bag of tasks for more work.

Discussion. Master/worker algorithms have good scalability as long as the number of tasks greatly exceeds the number of workers and the costs of the individual tasks are not so variable that some workers take drastically longer than the others.

Management of the bag of tasks can require global communication, and the overhead for this can limit efficiency. This effect is not a problem when the work associated with the tasks on average is much greater than the time required for management. In some cases, the designer might need to increase the size of each task to decrease the number of times the global task bag needs to be accessed.

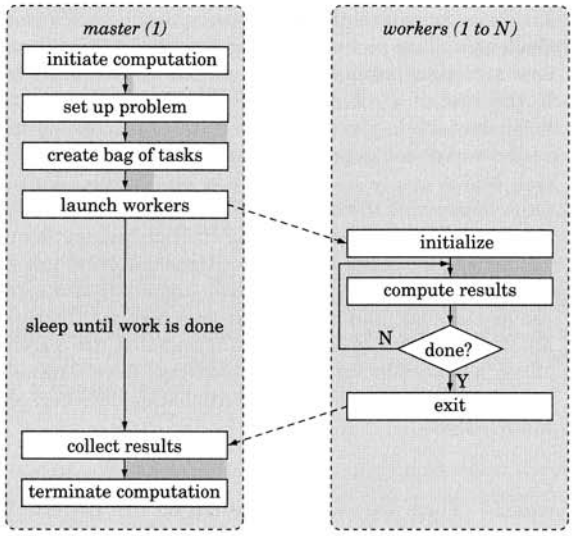

Figure 5.14: The two elements of the *Master/Worker* pattern are the *master* and the *worker*. There is only one master, but there can be one or more workers. Logically, the master sets up the calculation and then manages a bag of tasks. Each worker grabs a task from the bag, carries out the work, and then goes back to the bag, repeating until the termination condition is met.

The *Master/Worker* pattern is not tied to any particular hardware environment. Programs using this pattern work well on everything from clusters to SMP machines. It is, of course, beneficial if the programming environment provides support for managing the bag of tasks.

Detecting completion. One of the challenges in working with master/worker programs is to correctly determine when the entire problem is complete. This needs to be done in a way that is efficient but also guarantees that all of the work is complete before workers shut down.

- In the simplest case, all tasks are placed in the bag before the workers begin. Then each task continues until the bag is empty, at which point the workers terminate.
- Another approach is to use a queue to implement the task bag and arrange for the master or a worker to check for the desired termination condition. When it is detected, a *poison pill*, a special task that tells the workers to terminate, is created. The poison pill must be placed in the bag in such a way that it will be picked up on the next round of work. Depending on how the set of shared tasks are managed, it may be necessary to create one poison pill for each remaining worker to ensure that all workers receive the termination condition.

- Problems for which the set of tasks is not known initially produce unique challenges. This occurs, for example, when workers can add tasks as well as consume them (such as in applications of the *Divide and Conquer* pattern). In this case, it is not necessarily true that when a worker finishes a task and finds the task bag empty that there is no more work to do—another still-active worker could generate a new task. One must therefore ensure that the task bag is empty *and* all workers are finished. Further, in systems based on asynchronous message passing, it must be determined that there are no messages in transit that could, on their arrival, result in the creation of a new task. There are many known algorithms that solve this problem. For example, suppose the tasks are conceptually organized into a tree, where the root is the master task, and the children of a task are the tasks it generates. When all of the children of a task have terminated, the parent task can terminate. When all the children of the master task have terminated, the computation has terminated. Algorithms for termination detection are described in [BT89, Mat87, DS80].

Variations. There are several variations on this pattern. Because of the simple way it implements dynamic load balancing, this pattern is very popular, especially in embarrassingly parallel problems (as described in the *Task Parallelism* pattern). Here are a few of the more common variations.

- The master may turn into a worker after it has created the tasks. This is an effective technique when the termination condition can be detected without explicit action by the master (that is, the tasks can detect the termination condition on their own from the state of the bag of tasks).
- When the concurrent tasks map onto a simple loop, the master can be implicit and the pattern can be implemented as a loop with dynamic iteration assignment as described in the *Loop Parallelism* pattern.
- A centralized task queue can become a bottleneck, especially in a distributed-memory environment. An optimal solution [FLR98] is based on random work stealing. In this approach, each PE maintains a separate double-ended task queue. New tasks are placed in the front of the task queue of the local PE. When a task is completed, a subproblem is removed from the front of the local task queue. If the local task queue is empty, then another PE is chosen randomly, and a subproblem from the back of its task queue is "stolen". If that queue is also empty, then the PE tries again with another randomly chosen PE. This is particularly effective when used in problems based on the *Divide and Conquer* pattern. In this case, the tasks at the back of the queue were inserted earlier and hence represent larger subproblems. Thus, this approach tends to move large subproblems while handling the finer-grained subproblems at the PE where they were created. This helps the load balance and reduces overhead for the small tasks created in the deeper levels of recursion.

- The *Master/Worker* pattern can be modified to provide a modest level of fault tolerance [BDK95]. The master maintains two queues: one for tasks that still need to be assigned to workers and another for tasks that have already been assigned, but not completed. After the first queue is empty, the master can redundantly assign tasks from the "not completed" queue. Hence, if a worker dies and therefore can't complete its tasks, another worker will cover the unfinished tasks.

Examples

We will start with a generic description of a simple master/worker problem and then provide a detailed example of using the *Master/Worker* pattern in the parallel implementation of a program to generate the Mandelbrot set. Also see the Examples section of the *Shared Queue* pattern, which illustrates the use of shared queues by developing a master/worker implementation of a simple Java framework for programs using the *Fork/Join* pattern.

Generic solutions. The key to the master/worker program is the structure that holds the bag of tasks. The code in this section uses a task queue. We implement the task queue as an instance of the *Shared Queue* pattern.

The master process, shown in Fig. 5.15, initializes the task queue, representing each task by an integer. It then uses the *Fork/Join* pattern to create the worker

```
Int const Ntasks // Number of tasks
Int const Nworkers // Number of workers

SharedQueue :: task_queue; // task queue
SharedQueue :: global_results; // queue to hold results

void master()
{
   void worker()

   // Create and initialize shared data structures
   task_queue = new SharedQueue()
   global_results = new SharedQueue()

   for (int i = 0; i < N; i++)
      enqueue(task_queue, i)

   // Create Nworkers threads executing function Worker()
   ForkJoin (Nworkers, Worker)

   consume_the_results (Ntasks)
}
```

Figure 5.15: Master process for a master/worker program. This assumes a shared address space so the task and results queues are visible to all UEs. In this simple version, the master initializes the queue, launches the workers, and then waits for the workers to finish (that is, the `ForkJoin` command launches the workers and then waits for them to finish before returning). At that point, results are consumed and the computation completes.

```
void worker()
  {
    Int :: i
    Result :: res

    while (!empty(task_queue) {
       i = dequeue(task_queue)
       res = do_lots_of_work(i)
       enqueue(global_results, res)
    }
  }
```

Figure 5.16: Worker process for a master/worker program. We assume a shared address space thereby making `task_queue` and `global_results` available to the master and all workers. A worker loops over the `task_queue` and exits when the end of the queue is encountered.

processes or threads and wait for them to complete. When they have completed, it consumes the results.

The worker, shown in Fig. 5.16, loops until the task queue is empty. Every time through the loop, it takes the next task and does the indicated work, storing the results in a global results queue. When the task queue is empty, the worker terminates.

Note that we ensure safe access to the key shared variables (`task_queue` and `global_results`) by using instances of the *Shared Queue* pattern.

For programs written in Java, a thread-safe queue can be used to hold `Runnable` objects that are executed by a set of threads whose `run` methods behave like the worker threads described previously: removing a `Runnable` object from the queue and executing its `run` method. The `Executor` interface in the `java.util.concurrent` package in Java 2 1.5 provides direct support for the *Master/Worker* pattern. Classes implementing the interface provide an `execute` method that takes a `Runnable` object and arranges for its execution. Different implementations of the `Executor` interface provide different ways of managing the `Thread` objects that actually do the work. The `ThreadPoolExecutor` implements the *Master/Worker* pattern by using a fixed pool of threads to execute the commands. To use `Executor`, the program instantiates an instance of a class implementing the interface, usually using a factory method in the `Executors` class. For example, the code in Fig. 5.17 sets up a `ThreadPoolExecutor` that creates `num_threads` threads. These threads execute tasks specified by `Runnable` objects that are placed in an unbounded queue.

After the `Executor` has been created, a `Runnable` object whose `run` method specifies the behavior of the task can be passed to the `execute` method, which

```
/*create a ThreadPoolExecutor with an unbounded queue*/
Executor exec = new Executors.newFixedThreadPool(num_threads);
```

Figure 5.17: Instantiating and initializing a pooled executor

arranges for its execution. For example, assume the `Runnable` object is referred to by a variable `task`. Then for the executor defined previously, `exec.execute(task);` will place the task in the queue, where it will eventually be serviced by one of the executor's worker threads.

The *Master/Worker* pattern can also be used with SPMD programs and MPI. Maintaining the global queues is more challenging, but the overall algorithm is the same. A more detailed description of using MPI for shared queues appears in the *Implementation Mechanisms* design space.

Mandelbrot set generation. Generating the Mandelbrot set is described in detail in the Examples section of the *SPMD* pattern. The basic idea is to explore a quadratic recurrence relation at each point in a complex plane and color the point based on the rate at which the recursion converges or diverges. Each point in the complex plane can be computed independently and hence the problem is embarrassingly parallel (see the *Task Parallelism* pattern).

In Fig. 5.18, we reproduce the pseudocode given in the *SPMD* pattern for a sequential version of this problem. The program loops over the rows of the image displaying one row at a time as they are computed.

On homogeneous clusters or lightly-loaded shared-memory multiprocessor computers, approaches based on the *SPMD* or *Loop Parallelism* patterns are most effective. On a heterogeneous cluster or a multiprocessor system shared among many users (and hence with an unpredictable load on any given PE at any given time), a master/worker approach will be more effective.

We will create a master/worker version of a parallel Mandelbrot program based on the high-level structure described earlier. The master will be responsible for graphing the results. In some problems, the results generated by the workers interact and it can be important for the master to wait until all the workers have

```
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map

Real :: conv // divergence rate for a pixel
Array of Int :: color_map (M) // pixel color based on Conv rate
Array of Int :: row (RowSize) // Pixels to draw
Array of real :: ranges(2) // ranges in X and Y dimensions

manage_user_input(ranges, color_map) // input ranges, color map
initialize_graphics(RowSize, Nrows, M, ranges, color_map)

for (int i = 0; i<Nrows; i++){

    compute_Row (RowSize, ranges, row)

    graph(i, RowSize, M, color_map, ranges, row)

} // end loop [i] over rows
```

Figure 5.18: Pseudocode for a sequential version of the Mandelbrot set generation program

completed before consuming results. In this case, however, the results do not interact, so we split the fork and join operations and have the master plot results as they become available. Following the `Fork`, the master must wait for results to appear on the `global_results` queue. Because we know there will be one result per row, the master knows in advance how many results to fetch and the termination condition is expressed simply in terms of the number of iterations of the loop. After all the results have been plotted, the master waits at the `Join` function until all the workers have completed, at which point the master completes. Code is shown in Fig. 5.19.

```
Int const Ntasks // number of tasks
Int const Nworkers // number of workers
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map
typedef Row :: struct of {
     int :: index
     array of int :: pixels (RowSize)
} temp_row;
Array of Int :: color_map (M) // pixel color based on conv rate
Array of Real :: ranges(2) // ranges in X and Y dimensions
SharedQueue of Int :: task_queue; // task queue
SharedQueue of Row :: global_results; // queue to hold results

void master()
{
   void worker();

   manage_user_input(ranges, Color_map) // input ranges, color map
   initialize_graphics(RowSize, Nrows, M, ranges, color_map)

   // Create and initialize shared data structures
   task_queue = new SharedQueue();
   global_results = new SharedQueue();

   for (int i = 0; i < Nrows; i++)
      enqueue(task_queue, i);

   // Create Nworkers threads executing function worker()
   Fork (Nworkers, worker);

   // Wait for results and graph them as they appear
   for (int i = 0; i< Nrows; i++) {
       while (empty(task_queue) { // wait for results
          wait
       }
       temp_row = dequeue(global_results)
       graph(temp_row.index, RowSize, M, color_map, ranges, Row.pixels)
   }

   // Terminate the worker UEs
   Join (Nworkers);
}
```

Figure 5.19: Master process for a master/worker parallel version of the Mandelbrot set generation program


```
void worker()
{
  Int i, irow;
  Row temp_row;

  while (!empty(task_queue) {
     irow = dequeue(task_queue);
     compute_Row (RowSize, ranges, irow, temp_row.pixels)
     temp_row.index = irow
     enqueue(global_results, temp_row);
  }
}
```

Figure 5.20: Worker process for a master/worker parallel version of the Mandelbrot set generation program. We assume a shared address space thereby making `task_queue`, `global_results`, and `ranges` available to the master and the workers.

Notice that this code is similar to the generic case discussed earlier, except that we have overlapped the processing of the results with their computation by splitting the `Fork` and `Join`. As the names imply, `Fork` launches UEs running the indicated function and `Join` causes the master to wait for the workers to cleanly terminate. See the *Shared Queue* pattern for more details about the queue.

The code for the worker is much simpler and is shown in Fig 5.20. First, note that we assume the shared variables such as the queues and computation parameters are globally visible to the master and the workers. Because the queue is filled by the master before forking the workers, the termination condition is simply given by an empty queue. Each worker grabs a row index, does the computation, packs the row index and the computed row into the result queue, and continues until the queue is empty.

Known uses. This pattern is extensively used with the Linda programming environment. The tuple space in Linda is ideally suited to programs that use the *Master/Worker* pattern, as described in depth in [CG91] and in the survey paper [CGMS94].

The *Master/Worker* pattern is used in many distributed computing environments because these systems must deal with extreme levels of unpredictability in the availability of resources. The SETI@home project [SET] uses the *Master/Worker* pattern to utilize volunteers' Internet-connected computers to download and analyze radio telescope data as part of the Search for Extraterrestrial Intelligence (SETI). Programs constructed with the Calypso system [BDK95], a distributed computing framework which provides system support for dynamic changes in the set of PEs, also use the *Master/Worker* pattern. A parallel algorithm for detecting repeats in genomic data [RHB03] uses the *Master/Worker* pattern with MPI on a cluster of dual-processor PCs.

Related Patterns

This pattern is closely related to the *Loop Parallelism* pattern when the loops utilize some form of dynamic scheduling (such as when the `schedule(dynamic)` clause is used in OpenMP).

Implementations of the *Fork/Join* pattern sometimes use the *Master/Worker* pattern behind the scenes. This pattern is also closely related to algorithms that make use of the `nextval` function from TCGMSG [Har91, WSG95, LDSH95]. The `nextval` function implements a monotonic counter. If the bag of tasks can be mapped onto a fixed range of monotonic indices, the counter provides the bag of tasks and the function of the master is implied by the counter.

Finally, the owner-computes filter discussed in the molecular dynamics example in the *SPMD* pattern is essentially a variation on the master/worker theme. In such an algorithm, all the master would do is set up the bag of tasks (loop iterations) and assign them to UEs, with the assignment of tasks to UEs defined by the filter. Because the UEs can essentially perform this assignment themselves (by examining each task with the filter), no explicit master is needed.

5.6 THE *LOOP PARALLELISM* PATTERN

Problem

Given a serial program whose runtime is dominated by a set of computationally intensive loops, how can it be translated into a parallel program?

Context

The overwhelming majority of programs used in scientific and engineering applications are expressed in terms of iterative constructs; that is, they are loop-based. Optimizing these programs by focusing strictly on the loops is a tradition dating back to the older vector supercomputers. Extending this approach to modern parallel computers suggests a parallel algorithm strategy in which concurrent tasks are identified as iterations of *parallelized* loops.

The advantage of structuring a parallel algorithm around parallelized loops is particularly important in problems for which well-accepted programs already exist. In many cases, it isn't practical to massively restructure an existing program to gain parallel performance. This is particularly important when the program (as is frequently the case) contains convoluted code and poorly understood algorithms.

This pattern addresses ways to structure loop-based programs for parallel computation. When existing code is available, the goal is to "evolve" a sequential program into a parallel program by a series of transformations on the loops. Ideally, all changes are localized to the loops with transformations that remove loop-carried dependencies and leave the overall program semantics unchanged. (Such transformations are called *semantically neutral transformations*).

Not all problems can be approached in this loop-driven manner. Clearly, it will only work when the algorithm structure has most, if not all, of the computationally intensive work buried in a manageable number of distinct loops. Furthermore, the body of the loop must result in loop iterations that work well as parallel tasks (that is, they are computationally intensive, express sufficient concurrency, and are mostly independent).

Not all target computer systems align well with this style of parallel programming. If the code cannot be restructured to create effective distributed data

structures, some level of support for a shared address space is essential in all but the most trivial cases. Finally, Amdahl's law and its requirement to minimize a program's serial fraction often means that loop-based approaches are only effective for systems with smaller numbers of PEs.

Even with these restrictions, this class of parallel algorithms is growing rapidly. Because loop-based algorithms are the traditional approach in high-performance computing and are still dominant in new programs, there is a large backlog of loop-based programs that need to be ported to modern parallel computers. The OpenMP API was created primarily to support parallelization of these loop-driven problems. Limitations on the scalability of these algorithms are serious, but acceptable, given that there are orders of magnitude more machines with two or four processors than machines with dozens or hundreds of processors.

> *This pattern is particularly relevant for OpenMP programs running on shared-memory computers and for problems using the* Task Parallelism *and* Geometric Decomposition *patterns.*

Forces

- **Sequential equivalence.** A program that yields identical results (except for round-off errors) when executed with one thread or many threads is said to be *sequentially equivalent* (also known as *serially equivalent*). Sequentially equivalent code is easier to write, easier to maintain, and lets a single program source code work for serial and parallel machines.
- **Incremental parallelism (or refactoring).** When parallelizing an existing program, it is much easier to end up with a correct parallel program if (1) the parallelization is introduced as a sequence of incremental transformations, one loop at a time, and (2) the transformations don't "break" the program, allowing testing to be carried out after each transformation.
- **Memory utilization.** Good performance requires that the data access patterns implied by the loops mesh well with the memory hierarchy of the system. This can be at odds with the previous two forces, causing a programmer to massively restructure loops.

Solution

This pattern is closely aligned with the style of parallel programming implied by OpenMP. The basic approach consists of the following steps.

- **Find the bottlenecks.** Locate the most computationally intensive loops either by inspection of the code, by understanding the performance needs of each subproblem, or through the use of program performance analysis tools. The amount of total runtime on representative data sets contained by these loops will ultimately limit the scalability of the parallel program (see Amdahl's law).

- **Eliminate loop-carried dependencies.** The loop iterations must be nearly independent. Find dependencies between iterations or read/write accesses and transform the code to remove or mitigate them. Finding and removing the dependencies is discussed in the *Task Parallelism* pattern, while protecting dependencies with synchronization constructs is discussed in the *Shared Data* pattern.

- **Parallelize the loops.** Split up the iterations among the UEs. To maintain sequential equivalence, use semantically neutral directives such as those provided with OpenMP (as described in the OpenMP appendix, Appendix A). Ideally, this should be done to one loop at a time with testing and careful inspection carried out at each point to make sure race conditions or other errors have not been introduced.

- **Optimize the loop schedule.** The iterations must be scheduled for execution by the UEs so the load is evenly balanced. Although the right schedule can often be chosen based on a clear understanding of the problem, frequently it is necessary to experiment to find the optimal schedule.

This approach is only effective when the compute times for the loop iterations are large enough to compensate for parallel loop overhead. The number of iterations per loop is also important, because having many iterations per UE provides greater scheduling flexibility. In some cases, it might be necessary to transform the code to address these issues.

Two transformations commonly used are the following:

- **Merge loops.** If a problem consists of a sequence of loops that have consistent loop limits, the loops can often be merged into a single loop with more complex loop iterations, as shown in Fig. 5.21.

- **Coalesce nested loops.** Nested loops can often be combined into a single loop with a larger combined iteration count, as shown in Fig. 5.22. The larger number of iterations can help overcome parallel loop overhead, by (1) creating more concurrency to better utilize larger numbers of UEs, and (2) providing additional options for how the iterations are scheduled onto UEs.

Parallelizing the loops is easily done with OpenMP by using the `omp parallel for` directive. This directive tells the compiler to create a team of threads (the UEs in a shared-memory environment) and to split up loop iterations among the team. The last loop in Fig. 5.22 is an example of a loop parallelized with OpenMP. We describe this directive at a high level in the *Implementation Mechanisms* design space. Syntactic details are included in the OpenMP appendix, Appendix A.

Notice that in Fig. 5.22 we had to direct the system to create copies of the indices `i` and `j` local to each thread. The single most common error in using this pattern is to neglect to "privatize" key variables. If `i` and `j` are shared, then updates of `i` and `j` by different UEs can collide and lead to unpredictable results (that is, the program will contain a race condition). Compilers usually will not detect these errors, so programmers must take great care to make sure they avoid these situations.


```
#define N 20
#define Npoints 512

void FFT(); // a function to apply an FFT
void invFFT(); // a function to apply an inverse FFT
void filter(); // a frequency space filter
void setH(); // Set values of filter, H

int main() {
   int i, j;
   double A[Npoints], B[Npoints], C[Npoints], H[Npoints];

   setH(Npoints, H);

   // do a bunch of work resulting in values for A and C

   // method one: distinct loops to compute A and C
   for(i=0; i<N; i++){
      FFT (Npoints, A, B); // B = transformed A
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, A); // A = inv transformed B
   }
   for(i=0; i<N; i++){
      FFT (Npoints, C, B); // B = transformed C
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, C); // C = inv transformed B
   }

   // method two: the above pair of loops combined into
   // a single loop
   for(i=0; i<N; i++){
      FFT (Npoints, A, B); // B = transformed A
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, A); // A = inv transformed B
      FFT (Npoints, C, B); // B = transformed C
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, C); // C = inv transformed B
   }
   return 0;
}
```

Figure 5.21: Program fragment showing merging loops to increase the amount of work per iteration

The key to the application of this pattern is to use *semantically neutral* modifications to produce *sequentially equivalent* code. A semantically neutral modification doesn't change the meaning of the single-threaded program. Techniques for loop merging and coalescing of nested loops described previously, when used appropriately, are examples of semantically neutral modifications. In addition, most of the directives in OpenMP are semantically neutral. This means that adding the directive and running it with a single thread will give the same result as running the original program without the OpenMP directive.

Two programs that are semantically equivalent (when run with a single thread) need not both be sequentially equivalent. Recall that sequentially equivalent means

```
#define N 20
#define M 10

extern double work(); // a time-consuming function

int main() {

   int i, j, ij;
   double A[N][M];

   // method one: nested loops

   for(j=0; j<N; j++){
      for(i=0; i<M; i++){
         A[i][j] = work(i,j);
      }
   }

   // method two: the above pair of nested loops combined into
   // a single loop.

   for(ij=0; ij<N*M; ij++){
       j = ij/N;
       i = ij%M;

       A[i][j] = work(i,j);
   }

   // method three: the above loop parallelized with OpenMP.
   // The omp pragma creates a team of threads and maps
   // loop iterations onto them. The private clause
   // tells each thread to maintain local copies of ij, j, and i.

#pragma omp parallel for private(ij, j, i)
   for(ij=0; ij<N*M; ij++){
       j = ij/N;
       i = ij%M;

       A[i][j] = work(i,j);
   }
   return 0;
}
```

Figure 5.22: Program fragment showing coalescing nested loops to produce a single loop with a larger number of iterations

that the program will give the same result (subject to round-off errors due to changing the order of floating-point operations) whether run with one thread or many. Indeed, the (semantically neutral) transformations that eliminate loop-carried dependencies are motivated by the desire to change a program that is not sequentially equivalent to one that is. When transformations are made to improve performance, even though the transformations are semantically neutral, one must be careful that sequential equivalence has not been lost.

It is much more difficult to define sequentially equivalent programs when the code mentions either a thread ID or the number of threads. Algorithms that

reference thread IDs and the number of threads tend to favor particular threads or even particular numbers of threads, a situation that is dangerous when the goal is a sequentially equivalent program.

When an algorithm depends on the thread ID, the programmer is using the *SPMD* pattern. This may be confusing. SPMD programs can be loop-based. In fact, many of the examples in the *SPMD* pattern are indeed loop-based algorithms. But they are not instances of the *Loop Parallelism* pattern, because they display the hallmark trait of an SPMD program—namely, they use the UE ID to guide the algorithm.

Finally, we've assumed that a directive-based system such as OpenMP is available when using this pattern. It is possible, but clearly more difficult, to apply this pattern without such a directive-based programming environment. For example, in object-oriented designs, one can use the *Loop Parallelism* pattern by making clever use of anonymous classes with parallel iterators. Because the parallelism is buried in the iterators, the conditions of sequential equivalence can be met.

Performance considerations. In almost every application of this pattern, especially when used with OpenMP, the assumption is made that the program will execute on a computer that has multiple PEs sharing a single address space. This address space is assumed to provide equal-time access to every element of memory.

Unfortunately, this is usually not the case. Memories on modern computers are hierarchical. There are caches on the PEs, memory modules packaged with subsets of PEs, and other complications. While great effort is made in designing shared-memory multiprocessor computers to make them act like symmetric multiprocessor (SMP) computers, the fact is that all shared-memory computers display some degree of nonuniformity in memory access times across the system. In many cases, these effects are of secondary concern, and we can ignore how a program's memory access patterns match up with the target system's memory hierarchy. In other cases, particularly on larger shared-memory machines, programs must be explicitly organized according to the needs of the memory hierarchy. The most common trick is to make sure the data access patterns during initialization of key data structures match those during later computation using these data structures. This is discussed in more detail in [Mat03, NA01] and later in this pattern as part of the mesh computation example.

Another performance problem is false sharing. This occurs when variables are not shared between UEs, but happen to reside on the same cache line. Hence, even though the program semantics implies independence, each access by each UE requires movement of a cache line between UEs. This can create huge overheads as cache lines are repeatedly invalidated and moved between UEs as these supposedly independent variables are updated. An example of a program fragment that would incur high levels of false sharing is shown in Fig. 5.23. In this code, we have a pair of nested loops. The outermost loop has a small iteration count that will map onto the number of UEs (which we assume is four in this case). The innermost loop runs over a large number of time-consuming iterations. Assuming the iterations of the innermost loop are roughly equal, this loop should parallelize effectively.

```
#include <omp.h>
#define N 4 // Assume this equals the number of UEs
#define M 1000

extern double work(int, int); // a time-consuming function

int main() {

   int i, j;
   double A[N] = {0.0}; // Initialize the array to zero

// method one: a loop with false sharing from A since the elements
// of A are likely to reside in the same cache line.

#pragma omp parallel for private(j,i)
   for(j=0; j<N; j++){
      for(i=0; i<M; i++){
         A[j] += work(i,j);
      }
   }

// method two: remove the false sharing by using a temporary
// private variable in the innermost loop

   double temp;

#pragma omp parallel for private(j,i, temp)
   for(j=0; j<N; j++){
      temp = 0.0;
      for(i=0; i<M; i++){
         temp += work(i,j);
      }
      A[j] += temp;
   }
   return 0;
}
```

Figure 5.23: Program fragment showing an example of false sharing. The small array `A` is held in one or two cache lines. As the UEs access `A` inside the innermost loop, they will need to take ownership of the cache line back from the other UEs. This back-and-forth movement of the cache lines destroys performance. The solution is to use a temporary variable inside the innermost loop.

But the updates to the elements of the `A` array inside the innermost loop mean each update requires the UE in question to own the indicated cache line. Although the elements of `A` are truly independent between UEs, they likely sit in the same cache line. Hence, every iteration in the innermost loop incurs an expensive cache line invalidate-and-movement operation. It is not uncommon for this to not only destroy all parallel speedup, but to even cause the parallel program to become slower as more PEs are added. The solution is to create a temporary variable on each thread to accumulate values in the innermost loop. False sharing is still a factor, but only for the much smaller outermost loop where the performance impact is negligible.

Examples

As examples of this pattern in action, we will briefly consider the following:

- Numerical integration to estimate the value of a definite integral using the trapezoid rule
- Molecular dynamics, nonbonded energy computation
- Mandelbrot set computation
- Mesh computation

Each of these examples has been described elsewhere in detail. We will restrict our discussion in this pattern to the key loops and how they can be parallelized.

Numerical integration. Consider the problem of estimating the value of π using Eq. 5.5.

$$\pi = \int_0^1 \frac{4}{1 + x^2}\, dx \tag{5.5}$$

We use trapezoidal integration to numerically solve the integral. The idea is to fill the area under a curve with a series of rectangles. As the width of the rectangles approaches 0, the sum of the areas of the rectangles approaches the value of the integral.

A program to carry out this calculation on a single processor is shown in Fig. 5.24. To keep the program as simple as possible, we fix the number of steps to

```
#include <stdio.h>
#include <math.h>

int main () {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi %lf\n",pi);
   return 0;
}
```

Figure 5.24: Sequential program to carry out a trapezoid rule integration to compute $\int_0^1 \frac{4}{1+x^2}\, dx$

use in the integration at 1,000,000. The variable `sum` is initialized to 0 and the step size is computed as the range in `x` (equal to 1.0 in this case) divided by the number of steps. The area of each rectangle is the width (the step size) times the height (the value of the integrand at the center of the interval). The width is a constant, so we pull it out of the summation and multiply the sum of the rectangle heights by the step size, `step`, to get our estimate of the definite integral.

Creating a parallel version of this program using the *Loop Parallelism* pattern is simple. There is only one loop, so the inspection phase is trivial. To make the loop iterations independent, we recognize that (1) the values of the variable `x` are local to each iteration, so this variable can be handled as a thread-local or private variable and (2) the updates to `sum` define a reduction. Reductions are supported by the OpenMP API. Other than adding `#include <omp.h>`[2], only one additional line of code is needed to create a parallel version of the program. The following is placed above the `for` loop:

```
#pragma omp parallel for private(x) reduction(+:sum)
```

The pragma tells an OpenMP compiler to (1) create a team of threads, (2) create a private copy of `x` and `sum` for each thread, (3) initialize `sum` to 0 (the identity operand for addition), (4) map loop iterations onto threads in the team, (5) combine local values of `sum` into a single global value, and (6) join the parallel threads with the single master thread. Each of these steps is described in detail in the *Implementation Mechanisms* design space and the OpenMP appendix, Appendix A. For a non-OpenMP compiler, this pragma is ignored and therefore has no effect on the program's behavior.

Molecular dynamics. Throughout this book, we have used molecular dynamics as a recurring example. Molecular dynamics simulates the motions of a large molecular system. It uses an explicit time-stepping methodology where at each time step the force on each atom is computed and standard techniques from classical mechanics are used to compute how the forces change atomic motions.

The core algorithm, including pseudocode, was presented in Sec. 3.1.3 and in the *SPMD* pattern. The problem comes down to a collection of computationally expensive loops over the atoms within the molecular system. These are embedded in a top-level loop over time.

The loop over time cannot be parallelized because the coordinates and velocities from time step $t-1$ are the starting point for time step t. The individual loops over atoms, however, can be parallelized. The most important case to address is the nonbonded energy calculation. The code for this computation is shown in Fig. 5.25. Unlike the approach used in the examples from the *SPMD* pattern, we assume that the program and its data structures are unchanged from the serial case.

[2]The OpenMP include file defines function prototypes and opaque data types used by OpenMP.


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

    Int 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 5.25: Pseudocode for the nonbonded computation in a typical parallel molecular dynamics code. This is code is almost identical to the sequential version of the function shown previously in Fig. 4.4.

We will parallelize the `loop [i] over atoms`. Notice that the variables `forceX`, `forceY`, and `forceZ` are temporary variables used inside an iteration. We will need to create local copies of these private to each UE. The updates to the `force` arrays are reductions. Parallelization of this function would therefore require adding a single directive before the loop over atoms:

```
#pragma omp parallel for private(j, forceX, forceY, forceZ) \
                    reduction (+ : force)
```

The work associated with each atom varies unpredictably depending on how many atoms are in "its neighborhood". Although the compiler might be able to guess an effective schedule, in cases such as this one, it is usually best to try different schedules to find the one that works best. The work per atom is unpredictable, so one of the dynamic schedules available with OpenMP (and described in the OpenMP appendix, Appendix A) should be used. This requires the addition of a single `schedule` clause. Doing so gives us our final pragma for parallelizing this program:

```
#pragma omp parallel for private(j, forceX, forceY, forceZ) \
                reduction (+ : force) schedule (dynamic,10)
```

This schedule tells the compiler to group the loop iterations into blocks of size 10 and assign them dynamically to the UEs. The size of the blocks is arbitrary and chosen to balance dynamic scheduling overhead versus how effectively the load can be balanced.

OpenMP 2.0 for C/C++ does not support reductions over arrays so the reduction would need to be done explicitly. This is straightforward and is shown in Fig. 5.11. A future release of OpenMP will correct this deficiency and support reductions over arrays for all languages that support OpenMP.

The same method used to parallelize the nonbonded force computation could be used throughout the molecular dynamics program. The performance and scalability will lag the analogous SPMD version of the program. The problem is that each time a parallel directive is encountered, a new team of threads is in principle created. Most OpenMP implementations use a thread pool, rather than actually creating a new team of threads for each parallel region, which minimizes thread creation and destruction overhead. However, this method of parallelizing the computation still adds significant overhead. Also, the reuse of data from caches tends to be poor for these approaches. In principle, each loop can access a different pattern of atoms on each UE. This eliminates the capability for UEs to make effective use of values already in cache.

Even with these shortcomings, however, these approaches are commonly used when the goal is extra parallelism on a small shared-memory system [BBE+99]. For example, one might use an SPMD version of the molecular dynamics program across a cluster and then use OpenMP to gain extra performance from dual processors or from microprocessors utilizing simultaneous multithreading [MPS02].

Mandelbrot set computation. Consider the well-known Mandelbrot set [Dou86]. We discuss this problem and its parallelization as a task-parallel problem in the *Task Parallelism* and *SPMD* patterns. Each pixel is colored based on the behavior of the quadratic recurrence relation in Eq. 5.6.

$$Z_{n+1} = Z_n^2 + C \tag{5.6}$$

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 (-1.5 to 1.5) and the real part on the horizontal axis (-1 to 2). 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.

Pseudocode for the sequential version of this code is shown in Fig. 5.26. The interesting part of the problem is hidden inside the routine `compute_Row()`. The details of this routine are not important for understanding the parallel algorithm, however, so we will not show them here. At a high level, the following happens for each point in the row.

- Each pixel corresponds to a value of C in the quadratic recurrence. We compute this value based on the input `range` and the pixel indices.


```
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map

Real :: conv // divergence rate for a pixel
Array of Int :: color_map (M) // pixel color based on conv rate
Array of Int :: row (RowSize) // Pixels to draw
Array of Real :: ranges(2) // ranges in X and Y dimensions

manage_user_input(ranges, color_map) // input ranges, color map
initialize_graphics(RowSize, Nrows, M, ranges, color_map)

for (int i = 0; i<Nrows; i++){

    compute_Row (RowSize, ranges, row)

    graph(i, RowSize, M, color_map, ranges, row)

} // end loop [i] over rows
```

Figure 5.26: Pseudocode for a sequential version of the Mandelbrot set generation program

- We then compute the terms in the recurrence and set the value of the pixel based on whether it converges to a fixed value or diverges. If it diverges, we set the pixel value based on the rate of divergence.

Once computed, the rows are plotted to make the well-known Mandelbrot set images. The colors used for the pixels are determined by mapping divergence rates onto a color map.

Creating a parallel version of this program using the *Loop Parallelism* pattern is trivial. The iterations of the loop over rows are independent. All we need to do is make sure each thread has its own row to work on. We do this with the single pragma:

```
#pragma omp parallel for private(row)
```

The scheduling can be a bit tricky because work associated with each row will vary considerably depending on how many points diverge. The programmer should try several different schedules, but a cyclic distribution is likely to provide an effective load balance. In this schedule, the loop iterations are dealt out like a deck of cards. By interleaving the iterations among a set of threads, we are likely to get a balanced load. Because the scheduling decisions are static, the overhead incurred by this approach is small.

```
#pragma omp parallel for private(Row) schedule(static, 1)
```

For more information about the `schedule` clause and the different options available to the parallel programmer, see the OpenMP appendix, Appendix A.

Notice that we have assumed that the graphics package is thread-safe. This means that multiple threads can simultaneously call the library without causing any problems. The OpenMP specifications require this for the standard I/O library, but not for any other libraries. Therefore, it may be necessary to protect the call to the `graph` function by placing it inside a critical section:

```
#pragma critical
      graph(i, RowSize, M, color_map, ranges, row)
```

We describe this construct in detail in the *Implementation Mechanisms* design space and in the OpenMP appendix, Appendix A. This approach would work well, but it could have serious performance implications if the rows took the same time to compute and the threads all tried to graph their rows at the same time.

Mesh computation. Consider a simple mesh computation that solves the 1D heat-diffusion equation. The details of this problem and its solution using OpenMP are presented in the *Geometric Decomposition* pattern. We reproduce this solution in Fig. 5.27.

This program would work well on most shared-memory computers. A careful analysis of the program performance, however, would expose two performance problems. First, the `single` directive required to protect the swapping of the shared pointers adds an extra barrier, thereby greatly increasing the synchronization overhead. Second, on NUMA computers, memory access overhead is likely to be high because we've made no effort to keep the arrays near the PEs that will be manipulating them.

We address both of these problems in Fig. 5.28. To eliminate the need for the `single` directive, we modify the program so each thread has its own copy of the pointers `uk` and `ukp1`. This can be done with a `private` clause, but to be useful, we need the new, private copies of `uk` and `ukp1` to point to the shared arrays comprising the mesh of values. We do this with the `firstprivate` clause applied to the `parallel` directive that creates the team of threads.

The other performance issue we address, minimizing memory access overhead, is more subtle. As discussed earlier, to reduce memory traffic in the system, it is important to keep the data close to the PEs that will work with the data. On NUMA computers, this corresponds to making sure the pages of memory are allocated and "owned" by the PEs that will be working with the data contained in the page. The most common NUMA page-placement algorithm is the "first touch" algorithm, in which the PE first referencing a region of memory will have the page holding that memory assigned to it. So a very common technique in OpenMP programs is to initialize data in parallel using the same loop schedule as will be used later in the computations.


```
#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[]) {
    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;
    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 5.27: Parallel heat-diffusion program using OpenMP. This program is described in the Examples section of the *Geometric Decomposition* pattern.

We do this by first changing the initialization slightly so the initialization loop is identical to the computation loop. We then use the same loop parallelization directive on the initialization loop as on the computational loop. This doesn't guarantee an optimal mapping of memory pages onto the PEs, but it is a portable way to improve this mapping and in many cases come quite close to an optimal solution.

```
#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[]) {
    int i;
    uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
    ukp1[NX-1] = 0.0;
#pragma omp for schedule(static)
    for (i = 1; i < NX-1; ++i){
        uk[i] = 0.0;
        ukp1[i] = 0.0;
    }
}

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, temp) firstprivate(uk, ukp1)
    {
       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 */
          temp = ukp1; ukp1 = uk; uk = temp;
       }
    }
   return 0;
}
```

Figure 5.28: Parallel heat-diffusion program using OpenMP, with reduced thread management overhead and memory management more appropriate for NUMA computers

Known uses. The *Loop Parallelism* pattern is heavily used by OpenMP programmers. Annual workshops are held in North America (Wompat: Workshop on OpenMP Applications and Tools), Europe (EWOMP: European Workshop on OpenMP), and Japan (WOMPEI: Workshop on OpenMP Experiences and Implementations) to discuss OpenMP and its use. Proceedings from many of these workshops are widely available [VJKT00, Sci03, EV01] and are full of examples of the *Loop Parallelism* pattern.

Most of the work on OpenMP has been restricted to shared-memory multiprocessor machines for problems that work well with a nearly flat memory hierarchy. Work has been done to extend OpenMP applications to more complicated memory hierarchies, including NUMA machines [NA01, SSGF00] and even clusters [HLCZ99, SHTS01].

Related Patterns

The concept of driving parallelism from a collection of loops is general and used with many patterns. In particular, many problems using the *SPMD* pattern are loop-based. They use the UE ID, however, to drive the parallelization of the loop and hence don't perfectly map onto this pattern. Furthermore, problems using the *SPMD* pattern usually include some degree of parallel logic in between the loops. This allows them to decrease their serial fraction and is one of the reasons why SPMD programs tend to scale better than programs using the *Loop Parallelism* pattern.

Algorithms targeted for shared-memory computers that use the *Task Parallelism* or *Geometric Decomposition* patterns frequently use the *Loop Parallelism* pattern.

5.7 THE *FORK/JOIN* PATTERN

Problem

In some programs, the number of concurrent tasks varies as the program executes, and the way these tasks are related prevents the use of simple control structures such as parallel loops. How can a parallel program be constructed around such complicated sets of dynamic tasks?

Context

In some problems, the algorithm imposes a general and dynamic parallel control structure. Tasks are created dynamically (that is, *forked*) and later terminated (that is, *joined* with the forking task) as the program continues to execute. In most cases, the relationships between tasks are simple, and dynamic task creation can be handled with parallel loops (as described in the *Loop Parallelism* pattern) or through task queues (as described in the *Master/Worker* pattern). In other cases, relationships between the tasks within the algorithm must be captured in the way the tasks are managed. Examples include recursively generated task structures, highly irregular sets of connected tasks, and problems where different functions are mapped onto different concurrent tasks. In each of these examples, tasks are forked and later joined with the parent task (that is, the task that executed the fork) and the other tasks created by the same fork. These problems are addressed in the *Fork/Join* pattern.

As an example, consider an algorithm designed using the *Divide and Conquer* pattern. As the program execution proceeds, the problem is split into subproblems and new tasks are recursively created (or forked) to concurrently execute subproblems; each of these tasks may in turn be further split. When all the tasks created

to handle a particular split have terminated and joined with the parent task, the parent task continues the computation.

> *This pattern is particularly relevant for Java programs running on shared-memory computers and for problems using the* Divide and Conquer *and* Recursive Data *patterns. OpenMP can be used effectively with this pattern when the OpenMP environment supports nested parallel regions.*

Forces

- Algorithms imply relationships between tasks. In some problems, there are complex or recursive relations between tasks, and these relations need to be created and terminated dynamically. Although these can be mapped onto familiar control structures, the design in many cases is much easier to understand if the structure of the tasks is mimicked by the structure of the UEs.
- A one-to-one mapping of tasks onto UEs is natural in these algorithms, but that must be balanced against the number of UEs a system can handle.
- UE creation and destruction are costly operations. The algorithm might need to be recast to decrease these operations so they don't adversely affect the program's overall performance.

Solution

In problems that use the *Fork/Join* pattern, tasks map onto UEs in different ways. We will discuss two different approaches to the solution: (1) a simple direct mapping where there is one task per UE, and (2) an indirect mapping where a pool of UEs work on sets of tasks.

Direct task/UE mapping. The simplest case is one where we map each subtask to a UE. As new subtasks are forked, new UEs are created to handle them. This will build up corresponding sets of tasks and UEs. In many cases, there is a synchronization point where the main task waits for its subtasks to finish. This is called a *join*. After a subtask terminates, the UE handling it will be destroyed. We will provide an example of this approach later using Java.

The direct task/UE mapping solution to the *Fork/Join* pattern is the standard programming model in OpenMP. A program begins as a single thread (the *master thread*). A parallel construct forks a team of threads, the threads execute within a shared address space, and at the end of the parallel construct, the threads join back together. The original master thread then continues execution until the end of the program or until the next parallel construct.[3] This structure underlies

[3] In principle, nested parallel regions in OpenMP programs also map onto this direct-mapping solution. This approach has been successfully used in [AML+99]. The OpenMP specification, however, lets conforming OpenMP implementations "serialize" nested parallel regions (that is, execute them with a team of size one). Therefore, an OpenMP program cannot depend on nested parallel regions actually forking additional threads, and programmers must be cautious when using OpenMP for all but the simplest fork/join programs.

the implementation of the OpenMP parallel loop constructs described in the *Loop Parallelism* pattern.

Indirect task/UE mapping. Thread and process creation and destruction is one of the more expensive operations that occur in parallel programs. Thus, if a program contains repeated fork and join sections, the simple solution, which would require repeated destruction and creation of UEs, might not be efficient enough. Also, if at some point there are many more UEs than PEs, the program might incur unacceptable overhead due to the costs of context switches.

In this case, it is desirable to avoid the dynamic UE creation by implementing the fork/join paradigm using a thread pool. The idea is to create a (relatively) static set of UEs before the first fork operation. The number of UEs is usually the same as the number of PEs. The mapping of tasks to UEs then occurs dynamically using a task queue. The UEs themselves are not repeatedly created and destroyed, but simply mapped to dynamically created tasks as the need arises. This approach, although complicated to implement, usually results in efficient programs with good load balance. We will discuss a Java program that uses this approach in the Examples section of this pattern.

In OpenMP, there is some controversy over the best approach to use with this indirect-mapping approach [Mat03]. An approach gaining credibility is one based on a new, proposed OpenMP workshare construct called a `taskqueue` [SHPT00]. The proposal actually defines two new constructs: a `taskqueue` and a `task`. As the name implies, the programmer uses a `taskqueue` construct to create the task queue. Inside the `taskqueue` construct, a `task` construct defines a block of code that will be packaged into a task and placed on the task queue. The team of threads (as usually created with a parallel construct), playing the role of a thread pool, pulls tasks off the queue and executes them until the queue is empty.

Unlike OpenMP parallel regions, `taskqueues` can be dependably nested to produce a hierarchy of task queues. The threads work across task queues using work-stealing to keep all threads fully occupied until all of the queues are empty. This approach has been shown to work well [SHPT00] and is likely to be adopted in a future OpenMP specification.

Examples

As examples of this pattern, we will consider direct-mapping and indirect-mapping implementations of a parallel mergesort algorithm. The indirect-mapping solution makes use of a Java package FJTasks [Lea00b]. The Examples section of the *Shared Queue* pattern develops a similar, but simpler, framework.

Mergesort using direct mapping. As an example, consider the straightforward implementation of a method to performs a mergesort in Java shown in Fig. 5.29. The method takes a reference to the array to be sorted and sorts the elements with indices ranging from `lo` (inclusive) to `hi` (exclusive). Sorting the entire array `A` is done by invoking `sort(A,0,A.length)`.

```
static void sort(final int[] A,final int lo, final int hi)
{ int n = hi - lo;

   //if not large enough to do in parallel, sort sequentially
   if (n <= THRESHOLD){ Arrays.sort(A,lo,hi); return; }
   else
   { //split array
      final int pivot = (hi+lo)/2;

      //create and start new thread to sort lower half
      Thread t = new Thread()
      { public void run()
         { sort(A, lo, pivot); }
      };
      t.start();

      //sort upper half in current thread
      sort(A,pivot,hi);

      //wait for other thread
      try{t.join();}
      catch (InterruptedException e){Thread.dumpStack();}

      //merge sorted arrays
      int[] ws = new int[n];
      System.arraycopy(A,lo,ws,0,n);
      int wpivot = pivot - lo;
      int wlo = 0;
      int whi = wpivot;
      for (int i = lo; i != hi; i++)
      { if((wlo < wpivot) && (whi >= n || ws[wlo] <= ws[whi]))
         { A[i] = ws[wlo++]; }
         else { A[i] = ws[whi++]; }
      }
   }
}
```

Figure 5.29: Parallel mergesort where each task corresponds to a thread

The first step of the method is to compute the size of the segment of the array to be sorted. If the size of the problem is too small to make the overhead of sorting it in parallel worthwhile, then a sequential sorting algorithm is used (in this case, the tuned quicksort implementation provided by the `Arrays` class in the `java.util` package). If the sequential algorithm is not used, then a pivot point is computed to divide the segment to be sorted. A new thread is forked to sort the lower half of the array, while the parent thread sorts the upper half. The new task is specified by the `run` method of an anonymous inner subclass of the `Thread` class. When the new thread has finished sorting, it terminates. When the parent thread finishes sorting, it performs a join to wait for the child thread to terminate and then merges the two sorted segments together.

This simple approach may be adequate in fairly regular problems where appropriate threshold values can easily be determined. We stress that it is crucial that the threshold value be chosen appropriately: If too small, the overhead from too many UEs can make the program run even slower than a sequential version. If too large, potential concurrency remains unexploited.


```
int groupSize = 4; //number of threads
FJTaskRunnerGroup group = new FJTaskRunnerGroup(groupSize);
group.invoke(new FJTask()
     { public void run()
        { synchronized(this)
           { sort(A,0, A.length); }

        }
     });
```

Figure 5.30: Instantiating `FJTaskRunnerGroup` and invoking the master task

Mergesort using indirect mapping. This example uses the FJTask framework included as part of the public-domain package `EDU.oswego.cs.dl.util.concurrent` [Lea00b].[4] Instead of creating a new thread to execute each task, an instance of a (subclass of) `FJTask` is created. The package then dynamically maps the `FJTask` objects to a static set of threads for execution. Although less general than a `Thread`, an `FJTask` is a much lighter-weight object than a thread and is thus much cheaper to create and destroy. In Fig. 5.30 and Fig. 5.31, we show how to modify the mergesort example to use `FJTasks` instead of Java threads. The needed classes are imported from package `EDU.oswego.cs.dl.util.concurrent`. Before starting any `FJTasks`, a `FJTaskRunnerGroup` must be instantiated, as shown in Fig. 5.30. This creates the threads that will constitute the thread pool and takes the number of threads (group size) as a parameter. Once instantiated, the master task is invoked using the `invoke` method on the `FJTaskRunnerGroup`.

The sort routine itself is similar to the previous version except that the dynamically created tasks are implemented by the `run` method of an `FJTask` subclass instead of a `Thread` subclass. The `fork` and `join` methods of `FJTask` are used to fork and join the task in place of the `Thread` `start` and `join` methods. Although the underlying implementation is different, from the programmer's viewpoint, this indirect method is very similar to the direct implementation shown previously.

A more sophisticated parallel implementation of mergesort is provided with the `FJTask` examples in the `util.concurrent` distribution. The package also includes functionality not illustrated by this example.

Known uses. The documentation with the FJTask package includes several applications that use the *Fork/Join* pattern. The most interesting of these include Jacobi iteration, a parallel divide-and-conquer matrix multiplication, a standard

[4] This package was the basis for the new facilities to support concurrency introduced via JSR166 in Java 2 1.5. Its author, Doug Lea, was a lead in the JSR effort. The FJTask framework is not part of Java 2 1.5, but remains available in [Lea00b].

```
static void sort(final int[] A,final int lo, final int hi) {
   int n = hi - lo;
   if (n <= THRESHOLD){ Arrays.sort(A,lo,hi); return; }
   else {
      //split array
      final int pivot = (hi+lo)/2;

      //override run method in FJTask to execute run method
      FJTask t = new FJTask()
      { public void run()
         { sort(A, lo, pivot); }
      };

      //fork new task to sort lower half of array
      t.fork();

      //perform sort on upper half in current task
      sort(A,pivot,hi);

      //join with forked task
      t.join();

      //merge sorted arrays as before, code omitted
   }
}
```

Figure 5.31: Mergesort using the FJTask framework

parallel-processing benchmark program that simulates heat diffusion across a mesh, LU matrix decomposition, integral computation using recursive Gaussian Quadrature, and an adaptation of the Microscope game.[5]

Because OpenMP is based on a fork/join programming model, one might expect heavy use of the *Fork/Join* pattern by OpenMP programmers. The reality is, however, that most OpenMP programmers use either the *Loop Parallelism* or *SPMD* patterns because the current OpenMP standard provides poor support for true nesting of parallel regions. One of the few published accounts of using the *Fork/Join* pattern with standard OpenMP is a paper where nested parallelism was used to provide fine-grained parallelism in an implementation of LAPACK [ARv03].

Extending OpenMP so it can use the *Fork/Join* pattern in substantial applications is an active area of research. We've mentioned one of these lines of investigation for the case of the indirect-mapping solution of the *Fork/Join* pattern (the task queue [SHPT00]). Another possibility is to support nested parallel regions with explicit groups of threads for the direct-mapping solution of the *Fork/Join* pattern (the Nanos OpenMP compiler [GAM+00]).

[5] According to the documentation for this application, this is the game that is played while looking through the microscope in the laboratory in *The 7th Guest* (T7G; A CD-ROM game for PCs). It is a board game in which two players compete to fill spaces on the board with their tiles, something like Reversi or Othello.

Related Patterns

Algorithms that use the *Divide and Conquer* pattern use the *Fork/Join* pattern.

The *Loop Parallelism* pattern, in which threads are forked just to handle a single parallel loop, is an instance of the *Fork/Join* pattern.

The *Master/Worker* pattern, which in turn uses the *Shared Queue* pattern, can be used to implement the indirect-mapping solution.

5.8 THE *SHARED DATA* PATTERN

Problem

How does one explicitly manage shared data inside a set of concurrent tasks?

Context

Most of the *Algorithm Structure* patterns simplify the handling of shared data by using techniques to "pull" the shared data "outside" the set of tasks. Examples include replication plus reduction in the *Task Parallelism* pattern and alternating computation and communication in the *Geometric Decomposition* pattern. For certain problems, however, these techniques do not apply, thereby requiring that shared data be explicitly managed inside the set of concurrent tasks.

For example, consider the *phylogeny problem* from molecular biology, as described in [YWC+96]. A phylogeny is a tree showing relationships between organisms. The problem consists of generating large numbers of subtrees as potential solutions and then rejecting those that fail to meet the various consistency criteria. Different sets of subtrees can be examined concurrently, so a natural task definition in a parallel phylogeny algorithm would be the processing required for each set of subtrees. However, not all sets must be examined—if a set S is rejected, all supersets of S can also be rejected. Thus, it makes sense to keep track of the sets still to be examined *and* the sets that have been rejected. Given that the problem naturally decomposes into nearly independent tasks (one per set), the solution to this problem would use the *Task Parallelism* pattern. Using the pattern is complicated, however, by the fact that all tasks need both read and write access to the data structure of rejected sets. Also, because this data structure changes during the computation, we cannot use the replication technique described in the *Task Parallelism* pattern. Partitioning the data structure and basing a solution on this data decomposition, as described in the *Geometric Decomposition* pattern, might seem like a good alternative, but the way in which the elements are rejected is unpredictable, so any data decomposition is likely to lead to a poor load balance.

Similar difficulties can arise any time shared data must be explicitly managed inside a set of concurrent tasks. The common elements for problems that need the *Shared Data* pattern are (1) at least one data structure is accessed by multiple tasks in the course of the program's execution, (2) at least one task modifies the shared data structure, and (3) the tasks potentially need to use the modified value during the concurrent computation.

Forces

- The results of the computation must be correct for *any* ordering of the tasks that could occur during the computation.
- Explicitly managing shared data can incur parallel overhead, which must be kept small if the program is to run efficiently.
- Techniques for managing shared data can limit the number of tasks that can run concurrently, thereby reducing the potential scalability of an algorithm.
- If the constructs used to manage shared data are not easy to understand, the program will be more difficult to maintain.

Solution

Explicitly managing shared data can be one of the more error-prone aspects of designing a parallel algorithm. Therefore, a good approach is to start with a solution that emphasizes simplicity and clarity of abstraction and then try more complex solutions if necessary to obtain acceptable performance. The solution reflects this approach.

Be sure this pattern is needed. The first step is to confirm that this pattern is truly needed; it might be worthwhile to revisit decisions made earlier in the design process (the decomposition into tasks, for example) to see whether different decisions might lead to a solution that fits one of the *Algorithm Structure* patterns without the need to explicitly manage shared data. For example, if the *Task Parallelism* pattern is a good fit, it is worthwhile to review the design and see if dependencies can be managed by replication and reduction.

Define an abstract data type. Assuming this pattern must indeed be used, start by viewing the shared data as an abstract data type (ADT) with a fixed set of (possibly complex) operations on the data. For example, if the shared data structure is a queue (see the *Shared Queue* pattern), these operations would consist of *put* (enqueue), *take* (dequeue), and possibly other operations, such as a test for an empty queue or a test to see if a specified element is present. Each task will typically perform a sequence of these operations. These operations should have the property that if they are executed serially (that is, one at a time, without interference from other tasks), each operation will leave the data in a consistent state.

The implementation of the individual operations will most likely involve a sequence of lower-level actions, the results of which should not be visible to other UEs. For example, if we implemented the previously mentioned queue using a linked list, a "take" operation actually involves a sequence of lower-level operations (which may themselves consist of a sequence of even lower-level operations):

1. Use variable `first` to obtain a reference to the first object in the list.
2. From the first object, get a reference to the second object in the list.
3. Replace the value of `first` with the reference to the second object.

4. Update the size of the list.
5. Return the first element.

If two tasks are executing "take" operations concurrently, and these lower-level operations are interleaved (that is, the "take" operations are not being executed atomically), the result could easily be an inconsistent list.

Implement an appropriate concurrency-control protocol. After the ADT and its operations have been identified, the objective is to implement a concurrency-control protocol to ensure that these operations give the same results as if they were executed serially. There are several ways to do this; start with the first technique, which is the simplest, and then try the other more complex techniques if it does not yield acceptable performance. These more complex techniques can be combined if more than one is applicable.

One-at-a-time execution. The easiest solution is to ensure that the operations are indeed executed serially.

In a shared-memory environment, the most straightforward way to do this is to treat each operation as part of a single critical section and use a mutual-exclusion protocol to ensure that only one UE at a time is executing its critical section. This means that all of the operations on the data are mutually exclusive. Exactly how this is implemented will depend on the facilities of the target programming environment. Typical choices include mutex locks, synchronized blocks, critical sections, and semaphores. These mechanisms are described in the *Implementation Mechanisms* design space. If the programming language naturally supports the implementation of abstract data types, it is usually appropriate to implement each operation as a procedure or method, with the mutual-exclusion protocol implemented in the method itself.

In a message-passing environment, the most straightforward way to ensure serial execution is to assign the shared data structure to a particular UE. Each operation should correspond to a message type; other processes request operations by sending messages to the UE managing the data structure, which processes them serially.

In either environment, this approach is usually not difficult to implement, but it can be overly conservative (that is, it might disallow concurrent execution of operations that would be safe to execute simultaneously), and it can produce a bottleneck that negatively affects the performance of the program. If this is the case, the remaining approaches described in this section should be reviewed to see whether one of them can reduce or eliminate this bottleneck and give better performance.

Noninterfering sets of operations. One approach to improving performance begins by analyzing the interference between the operations. We say that operation A *interferes with* operation B if A writes a variable that B reads. Notice that an operation may interfere with itself, which would be a concern if more than one task executes the same operation (for example, more than one task executes "take" operations on a shared queue). It may be the case, for example, that the operations fall into two disjoint sets, where the operations in different sets do not

interfere with each other. In this case, the amount of concurrency can be increased by treating each of the sets as a different critical section. That is, within each set, operations execute one a time, but operations in different sets can proceed concurrently.

Readers/writers. If there is no obvious way to partition the operations into disjoint sets, consider the type of interference. It may be the case that some of the operations modify the data, but others only read it. For example, if operation A is a writer (both reading and writing the data) and operation B is a reader (reading, but not writing, the data), A interferes with itself and with B, but B does not interfere with itself. Thus, if one task is performing operation A, no other task should be able to execute either A or B, but any number of tasks should be able to execute B concurrently. In such cases, it may be worthwhile to implement a readers/writers protocol that will allow this potential concurrency to be exploited. The overhead of managing the readers/writers protocol is greater than that of simple mutex locks, so the length of the readers' computation should be long enough to make this overhead worthwhile. In addition, there should generally be a larger number of concurrent readers than writers.

The `java.util.concurrent` package provides read/write locks to support the readers/writers protocol. The code in Fig. 5.32 illustrates how these locks

```
class X {
  ReadWriteLock rw = new ReentrantReadWriteLock();
  // ...

  /*operation A is a writer*/
  public void A() throws InterruptedException {
    rw.writeLock().lock(); //lock the write lock
    try {
      // ... do operation A
    }
    finally {
      rw.writeLock().unlock(); //unlock the write lock
    }
  }

 /*operation B is a reader*/
 public void B() throws InterruptedException {
    rw.readLock().lock(); //lock the read lock
    try {
      // ... do operation B
    }
    finally {
      rw.readLock().unlock(); //unlock the read lock
    }
  }
}
```

Figure 5.32: Typical use of read/write locks. These locks are defined in the `java.util.concurrent.locks` package. Putting the unlock in the `finally` block ensures that the lock will be unlocked regardless of how the `try` block is exited (normally or with an exception) and is a standard idiom in Java programs that use locks rather than synchronized blocks.

are typically used: First instantiate a `ReadWriteLock`, and then obtain its read and write locks. `ReentrantReadWriteLock` is a class that implements the `ReadWriteLock` interface. To perform a read operation, the read lock must be locked. To perform a write operation, the write lock must be locked. The semantics of the locks are that any number of UEs can simultaneously hold the read lock, but the write lock is exclusive; that is, only one UE can hold the write lock, and if the write lock is held, no UEs can hold the read lock either.

Readers/writers protocols are discussed in [And00] and most operating systems texts.

Reducing the size of the critical section. Another approach to improving performance begins with analyzing the implementations of the operations in more detail. It may be the case that only part of the operation involves actions that interfere with other operations. If so, the size of the critical section can be reduced to that smaller part. Notice that this sort of optimization is very easy to get wrong, so it should be attempted only if it will give significant performance improvements over simpler approaches, *and* the programmer completely understands the interferences in question.

Nested locks. This technique is a sort of hybrid between two of the previous approaches, noninterfering operations and reducing the size of the critical section. Suppose we have an ADT with two operations. Operation `A` does a lot of work both reading and updating variable `x` and then reads and updates variable `y` in a single statement. Operation `B` reads and writes `y`. Some analysis shows that UEs executing `A` need to exclude each other, UEs executing `B` need to exclude each other, and because both operations read and update `y`, technically, `A` and `B` need to mutually exclude each other as well. However, closer inspection shows that the two operations are *almost* noninterfering. If it weren't for that single statement where `A` reads and updates `y`, the two operations could be implemented in separate critical sections that would allow one `A` and one `B` to execute concurrently. A solution is to use two locks, as shown in Fig. 5.33. `A` acquires and holds `lockA` for the entire operation. `B` acquires and holds `lockB` for the entire operation. `A` acquires `lockB` and holds it only for the statement updating `y`.

Whenever nested locking is used, the programmer should be aware of the potential for deadlocks and double-check the code. (The classic example of deadlock, stated in terms of the previous example, is as follows: `A` acquires `lockA` and `B` acquires `lockB`. `A` then tries to acquire `lockB` and `B` tries to acquire `lockA`. Neither operation can now proceed.) Deadlocks can be avoided by assigning a partial order to the locks and ensuring that locks are always acquired in an order that respects the partial order. In the previous example, we would define the order to be `lockA` < `lockB` and ensure that `lockA` is never acquired by a UE already holding `lockB`.

Application-specific semantic relaxation. Yet another approach is to consider partially replicating shared data (the software caching described in [YWC+96]) and perhaps even allowing the copies to be inconsistent if this can be done without affecting the results of the computation. For example, a distributed-memory solution to the phylogeny problem described earlier might give

```
class Y {
  Object lockA = new Object();
  Object lockB = new Object();

  void A()
  { synchronized(lockA)
    {
      ....compute....
      synchronized(lockB)
      { ....read and update y....
      }
    }
  }

  void B() throws InterruptedException
  { synchronized(lockB)
    { ...compute....
    }
  }
}
```

Figure 5.33: Example of nested locking using synchronized blocks with dummy objects `lockA` and `lockB`

each UE its own copy of the set of sets already rejected and allow these copies to be out of synch; tasks may do extra work (in rejecting a set that has already been rejected by a task assigned to a different UE), but this extra work will not affect the result of the computation, and it may be more efficient overall than the communication cost of keeping all copies in synch.

Review other considerations

Memory synchronization. Make sure memory is synchronized as required: Caching and compiler optimizations can result in unexpected behavior with respect to shared variables. For example, a stale value of a variable might be read from a cache or register instead of the newest value written by another task, or the latest value might not have been flushed to memory and thus would not be visible to other tasks. In most cases, memory synchronization is performed implicitly by higher-level synchronization primitives, but it is still necessary to be aware of the issue. Unfortunately, memory synchronization techniques are very platform-specific. In OpenMP, the flush directive can be used to synchronize memory explicitly; it is implicitly invoked by several other directives. In Java, memory is implicitly synchronized when entering and leaving a synchronized block, and, in Java 2 1.5, when locking and unlocking locks. Also, variables marked `volatile` are implicitly synchronized with respect to memory. This is discussed in more detail in the *Implementation Mechanisms* design space.

Task scheduling. Consider whether the explicitly managed data dependencies addressed by this pattern affect task scheduling. A key goal in deciding how to schedule tasks is good load balance; in addition to the considerations described in the *Algorithm Structure* pattern being used, one should also take into account that

tasks might be suspended waiting for access to shared data. It makes sense to try to assign tasks in a way that minimizes such waiting, or to assign multiple tasks to each UE in the hope that there will always be one task per UE that is not waiting for access to shared data.

Examples

Shared queues. The shared queue is a commonly used ADT and an excellent example of the *Shared Data* pattern. The *Shared Queue* pattern discusses concurrency-control protocols and the techniques used to achieve highly efficient shared-queue programs.

Genetic algorithm for nonlinear optimization. Consider the GAFORT program from the SPEC OMP2001 benchmark suite [ADE+01]. GAFORT is a small Fortran program (around 1,500 lines) that implements a genetic algorithm for nonlinear optimization. The calculations are predominantly integer arithmetic, and the program's performance is dominated by the cost of moving large arrays of data through the memory subsystem.

The details of the genetic algorithm are not important for this discussion. We are going to focus on a single loop within GAFORT. Pseudocode for the sequential version of this loop, based on the discussion of GAFORT in [EM], is shown in Fig. 5.34. This loop shuffles the population of chromosomes and consumes on the order of 36 percent of the runtime in a typical GAFORT job [AE03].

```
Int const NPOP // number of chromosomes (~40000)
Int const NCHROME // length of each chromosome

Real :: tempScalar
Array of Real :: temp(NCHROME)
Array of Int :: iparent(NCHROME, NPOP)
Array of Int :: fitness(NPOP)
Int :: j, iother

loop [j] over NPOP
   iother = rand(j) // returns random value greater
                    // than or equal to zero but not
                    // equal to j and less than NPOP

   // Swap Chromosomes
   temp(1:NCHROME) = iparent(1:NCHROME, iother)
   iparent(1:NCHROME, iother) = iparent(1:NCHROME, j)
   iparent(1:NCHROME, j) = temp(1:NCHROME)

   // Swap fitness metrics
   tempScalar = fitness(iother)
   fitness(iother) = fitness(j)
   fitness(j) = tempScalar

end loop [j]
```

Figure 5.34: Pseudocode for the population shuffle loop from the genetic algorithm program GAFORT

A parallel version of this program will be created by parallelizing the loop, using the *Loop Parallelism* pattern. In this example, the shared data consists of the `iparent` and `fitness` arrays. Within the body of the loop, calculations involving these arrays consist of swapping two elements of `iparent` and then swapping the corresponding elements of `fitness`. Examination of these operations shows that two swap operations interfere when at least one of the locations being swapped is the same in both operations.

Thinking about the shared data as an ADT helps us to identify and analyze the actions taken on the shared data. This does not mean, however, that the implementation itself always needs to reflect this structure. In some cases, especially when the data structure is simple and the programming language does not support ADTs well, it can be more effective to forgo the encapsulation implied in an ADT and work with the data directly. This example illustrates this.

As mentioned earlier, the chromosomes being swapped might interfere with each other; thus the loop over `j` cannot safely execute in parallel. The most straightforward approach is to enforce a "one at a time" protocol using a critical section, as shown in Fig. 5.35. It is also necessary to modify the random number generator

```
#include <omp.h>
   Int const NPOP // number of chromosomes (~40000)
   Int const NCHROME // length of each chromosome

   Real :: tempScalar
   Array of Real :: temp(NCHROME)
   Array of Int :: iparent(NCHROME, NPOP)
   Array of Int :: fitness(NPOP)
   Int :: j, iother

#pragma omp parallel for
   loop [j] over NPOP
      iother = par_rand(j) // returns random value greater
                           // than or equal to zero but not
                           // equal to j and less than NPOP

#pragma omp critical
{
      // Swap Chromosomes
      temp(1:NCHROME) = iparent(1:NCHROME, iother)
      iparent(1:NCHROME, iother) = iparent(1:NCHROME, j)
      iparent(1:NCHROME, j) = temp(1:NCHROME)

      // Swap fitness metrics
      tempScalar = fitness(iother)
      fitness(iother) = fitness(j)
      fitness(j) = tempScalar
}
   end loop [j]
```

Figure 5.35: Pseudocode for an ineffective approach to parallelizing the population shuffle in the genetic algorithm program GAFORT

so it produces a consistent set of pseudorandom numbers when called in parallel by many threads. The algorithms to accomplish this are well understood [Mas97], but will not be discussed here.

The program in Fig. 5.35 can safely execute with multiple threads, but it will not run any faster as more threads are added. In fact, this program will slow down as more threads are added because the threads will waste system resources as they wait for their turn to execute the critical section. In essence, the concurrency-control protocol eliminates all of the available concurrency.

The solution to this problem is to take advantage of the fact that the swap operations on the shared data only interfere when at least one of the locations being swapped is the same in both operations. Hence, the right concurrency-control protocol uses pairwise synchronization with nested locks, thereby adding only modest overhead when loop iterations do not interfere. The approach used in [ADE+01] is to create an OpenMP lock for each chromosome. Pseudocode for this solution is shown in Fig. 5.36. In the resulting program, most of the loop iterations do not actually interfere with each other. The total number of chromosomes, `NPOP` (40,000 in the SPEC OMP2001 benchmark), is much larger than the number of UEs, so there is only a slight chance that loop iterations will happen to interfere with another loop iteration.

OpenMP locks are described in the OpenMP appendix, Appendix A. The locks themselves use an opaque type, `omp_lock_t`, defined in the `omp.h` header file. The lock array is defined and later initialized in a separate parallel loop. Once inside the chromosome-swapping loop, the locks are set for the pair of swapping chromosomes, the swap is carried out, and the locks are unset. Nested locks are being used, so the possibility of deadlock must be considered. The solution here is to order the locks using the value of the indices of the array element associated with the lock. Always acquiring locks in this order will prevent deadlock when a pair of loop iterations happen to be swapping the same two elements at the same time. After the more efficient concurrency-control protocol is implemented, the program runs well in parallel.

Known uses. A solution to the phylogeny problem described in the Context section is presented in [YWC+96]. The overall approach fits the *Task Parallelism* pattern; the rejected-sets data structure is explicitly managed using replication and periodic updates to reestablish consistency among copies.

Another problem presented in [YWC+96] is the Gröbner basis program. Omitting most of the details, in this application the computation consists of using pairs of polynomials to generate new polynomials, comparing them against a master set of polynomials, and adding those that are not linear combinations of elements of the master set to the master set (where they are used to generate new pairs). Different pairs can be processed concurrently, so one can define a task for each pair and partition them among UEs. The solution described in [YWC+96] fits the *Task Parallelism* pattern (with a task queue consisting of pairs of polynomials), plus explicit management of the master set using an application-specific protocol called *software caching*.

```
#include <omp.h>
    Int const NPOP // number of chromosomes (~40000)
    Int const NCHROME // length of each chromosome

    Array of omp_lock_t :: lck(NPOP)

    Real :: tempScalar
    Array of Real :: temp(NCHROME)
    Array of Int :: iparent(NCHROME, NPOP)
    Array of Int :: fitness(NPOP)
    Int :: j, iother

// Initialize the locks
#pragma omp parallel for
    for (j=0; j<NPOP; j++){ omp_init_lock (&lck(j)) }

#pragma omp parallel for
    for (j=0; j<NPOP; j++){
        iother = par_rand(j) // returns random value >= 0, != j,
                             // < NPOP
        if (j < iother) {
           set_omp_lock (lck(j)); set_omp_lock (lck(iother))
        }
        else {
           set_omp_lock (lck(iother)); set_omp_lock (lck(j))
        }

        // Swap Chromosomes
        temp(1:NCHROME) = iparent(1:NCHROME, iother);
        iparent(1:NCHROME, iother) = iparent(1:NCHROME, j);
        iparent(1:NCHROME, j) = temp(1:NCHROME);

        // Swap fitness metrics
        tempScalar = fitness(iother)
        fitness(iother) = fitness(j)
        fitness(j) = tempScalar

        if (j < iother) {
           unset_omp_lock (lck(iother)); unset_omp_lock (lck(j))
        }
        else {
           unset_omp_lock (lck(j)); unset_omp_lock (lck(iother))
        }
    } // end loop [j]
```

Figure 5.36: Pseudocode for a parallelized loop to carry out the population shuffle in the genetic algorithm program GAFORT. This version of the loop uses a separate lock for each chromosome and runs effectively in parallel.

Related Patterns

The *Shared Queue* and *Distributed Array* patterns discuss specific types of shared data structures. Many problems that use the *Shared Data* pattern use the *Task Parallelism* pattern for the algorithm structure.

5.9 THE *SHARED QUEUE* PATTERN

Problem

How can concurrently-executing UEs safely share a queue data structure?

Context

Effective implementation of many parallel algorithms requires a queue that is to be shared among UEs. The most common situation is the need for a task queue in programs implementing the *Master/Worker* pattern.

Forces

- Simple concurrency-control protocols provide greater clarity of abstraction and make it easier for the programmer to verify that the shared queue has been correctly implemented.
- Concurrency-control protocols that encompass too much of the shared queue in a single synchronization construct increase the chances UEs will remain blocked waiting to access the queue and will limit available concurrency.
- A concurrency-control protocol finely tuned to the queue and how it will be used increases the available concurrency, but at the cost of much more complicated, and more error-prone, synchronization constructs.
- Maintaining a single queue for systems with complicated memory hierarchies (as found on NUMA machines and clusters) can cause excess communication and increase parallel overhead. Solutions may in some cases need to break with the single-queue abstraction and use multiple or distributed queues.

Solution

Ideally the shared queue would be implemented as part of the target programming environment, either explicitly as an ADT to be used by the programmer, or implicitly as support for the higher-level patterns (such as *Master/Worker*) that use it. In Java 2 1.5, such queues are available in the `java.util.concurrent` package. Here we develop implementations from scratch to illustrate the concepts.

Implementing shared queues can be tricky. Appropriate synchronization must be utilized to avoid race conditions, and performance considerations—especially for problems where large numbers of UEs access the queue—can require sophisticated synchronization. In some cases, a noncentralized queue might be needed to eliminate performance bottlenecks.

However, if it is necessary to implement a shared queue, it can be done as an instance of the *Shared Data* pattern: First, we design an ADT for the queue by defining the values the queue can hold and the set of operations on the queue. Next, we consider the concurrency-control protocols, starting with the simplest "one-at-a-time execution" solution and then applying a series of refinements. To make this discussion more concrete, we will consider the queue in terms of a specific problem:

a queue to hold tasks in a master/worker algorithm. The solutions presented here, however, are general and can be easily extended to cover other applications of a shared queue.

The abstract data type (ADT). An ADT is a set of values and the operations defined on that set of values. In the case of a queue, the values are ordered lists of zero or more objects of some type (for example, integers or task IDs). The operations on the queue are `put` (or *enqueue*) and `take` (or *dequeue*). In some situations, there might be other operations, but for the sake of this discussion, these two are sufficient.

We must also decide what happens when a `take` is attempted on an empty queue. What should be done depends on how termination will be handled by the master/worker algorithm. Suppose, for example, that all the tasks will be created at startup time by the master. In this case, an empty task queue will indicate that the UE should terminate, and we will want the `take` operation on an empty queue to return immediately with an indication that the queue is empty—that is, we want a *nonblocking* queue. Another possible situation is that tasks can be created dynamically and that UEs will terminate when they receive a special *poison-pill* task. In this case, appropriate behavior might be for the `take` operation on an empty queue to wait until the queue is nonempty—that is, we want a *block-on-empty* queue.

Queue with "one at a time" execution

Nonblocking queue. Because the queue will be accessed concurrently, we must define a concurrency-control protocol to ensure that interference by multiple UEs will not occur. As recommended in the *Shared Data* pattern, the simplest solution is to make all operations on the ADT exclude each other. Because none of the operations on the queue can block, a straightforward implementation of mutual exclusion as described in the *Implementation Mechanisms* design spaces suffices. The Java implementation shown in Fig. 5.37 uses a linked list to hold the tasks in the queue. (We develop our own list class rather than using an unsynchronized library class such as `java.util.LinkedList` or a class from the `java.util.concurrent` package to illustrate how to add appropriate synchronization.) `head` refers to an always-present dummy node.[6]

The first task in the queue (if any) is held in the node referred to by `head.next`. The `isEmpty` method is private, and only invoked inside a synchronized method. Thus, it need not be synchronized. (If it were public, it would need to be synchronized as well.) Of course, numerous ways of implementing the structure that holds the tasks are possible.

Block-on-empty queue. The second version of the shared queue is shown in Fig. 5.38. In this version of the queue, the `take` operation is changed so that

[6]The code for `take` makes the old head node into a dummy node rather than simply manipulating `next` pointers to allow us to later optimize the code so that `put` and `get` can execute concurrently.


```
public class SharedQueue1
{
   class Node //inner class defines list nodes
   { Object task;
     Node next;

     Node(Object task)
     {this.task = task; next = null;}
   }

   private Node head = new Node(null); //dummy node
   private Node last = head;

   public synchronized void put(Object task)
   { assert task != null: "Cannot insert null task";
     Node p = new Node(task);
     last.next = p;
     last = p;
   }

   public synchronized Object take()
   { //returns first task in queue or null if queue is empty
     Object task = null;
     if (!isEmpty())
     { Node first = head.next;
       task = first.task;
       first.task = null;
       head = first;
     }
     return task;
   }

   private boolean isEmpty(){return head.next == null;}
}
```

Figure 5.37: Queue that ensures that at most one thread can access the data structure at one time. If the queue is empty, `null` is immediately returned.

a thread trying to take from an empty queue will wait for a task rather than returning immediately. The waiting thread needs to release its lock and reacquire it before trying again. This is done in Java using the `wait` and `notify` methods. These are described in the Java appendix, Appendix C. The Java appendix also shows the queue implemented using locks from the `java.util.concurrent.locks` package introduced in Java 2 1.5 instead of `wait` and `notify`. Similar primitives are available with POSIX threads (Pthreads) [But97,IEE], and techniques for implementing this functionality with semaphores and other basic primitives can be found in [And00].

In general, to change a method that returns immediately if a condition is false to one that waits until the condition is true, two changes need to be made: First, we replace a statement of the form

```
if (condition){do_something;}
```

```
public class SharedQueue2
{
   class Node
   { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;

   public synchronized void put(Object task)
   { assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p;
      last = p;
      notifyAll();
   }

   public synchronized Object take()
   { //returns first task in queue, waits if queue is empty
      Object task = null;
      while (isEmpty())
         {try{wait();}catch(InterruptedException ignore){}}
      { Node first = head.next;
         task = first.task;
         first.task = null;
         head = first;
      }
      return task;
   }

   private boolean isEmpty(){return head.next == null;}
}
```

Figure 5.38: Queue that ensures at most one thread can access the data structure at one time. Unlike the first shared queue example, if the queue is empty, the thread waits. When used in a master/worker algorithm, a poison pill would be required to signal termination to a thread.

with a loop[7]

```
while( !condition){wait();} do_something;
```

Second, we examine the other operations on the shared queue and add a `notifyAll` to any operations that might establish `condition`. The result is an instance of

[7]The fact that `wait` can throw an `InterruptedException` must be dealt with; it is ignored here for clarity, but handled properly in the code examples.

the basic idiom for using `wait`, described in more detail in the Java appendix, Appendix C.

Thus, two major changes are made in moving to the code in Fig. 5.38. First, we replace the code

```
if (!isEmpty()){....}
```

with

```
while(isEmpty())
  {try{wait()}catch(InterruptedException ignore){}}{....}
```

Second, we note that the `put` method will make the queue not empty, so we add to it a call to `notifyAll`.

This implementation has a performance problem in that it will generate extraneous calls to `notifyAll`. This does not affect the correctness, but it might degrade the performance. One way this implementation could be optimized would be to minimize the number of invocations of `notifyAll` in `put`. One way to do this is to keep track of the number of waiting threads and only perform a `notifyAll` when there are threads waiting. We would have, for `int w` indicating the number of waiting threads:

```
while( !condition){w++; wait(); w--} do_something;
```

and

```
if (w>0) notifyAll();
```

In this particular example, because only one waiting thread will be able to consume a task, `notifyAll` could be replaced by `notify`, which notifies only one waiting thread. We show code for this refinement in a later example (Fig. 5.40).

Concurrency-control protocols for noninterfering operations. If the performance of the shared queue is inadequate, we must look for more efficient concurrency-control protocols. As discussed in the *Shared Data* pattern, we need to look for noninterfering sets of operations in our ADT. Careful examination of the operations in our nonblocking shared queue (see Fig. 5.37 and Fig. 5.38) shows that

```
public class SharedQueue3
{
   class Node
   { Object task;
     Node next;

     Node(Object task)
     {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;

   private Object putLock = new Object();
   private Object takeLock = new Object();

   public void put(Object task)
   { synchronized(putLock)
     { assert task != null: "Cannot insert null task";
       Node p = new Node(task);
       last.next = p;
       last = p;
     }
   }

   public Object take()
   { Object task = null;
     synchronized(takeLock)
     { if (!isEmpty())
       { Node first = head.next;
         task = first.task;
         first.task = null;
         head = first;
       }
     }
     return task;
   }
}
```

Figure 5.39: Shared queue that takes advantage of the fact that `put` and `take` are noninterfering and uses separate locks so they can proceed concurrently

the `put` and `take` are noninterfering because they do not access the same variables. The `put` method modifies the reference `last` and the `next` member of the object referred to by `last`. The `take` method modifies the value of the `task` member in the object referred to by `head.next` and the reference `head`. Thus, `put` modifies `last` and the `next` member of some `Node` object. The `take` method modifies `head` and the `task` member of some object. These are noninterfering operations, so we can use one lock for `put` and a different lock for `take`. This solution is shown in Fig. 5.39.

Concurrency-control protocols using nested locks. The approach shown in Fig. 5.39 isn't as easy to apply to a block-on-empty queue, however. First of all, the

`wait`, `notify`, and `notifyAll` methods on an object can only be invoked within a block synchronized on that object. Also, if we have optimized the invocations of `notify` as described previously, then `w`, the count of waiting threads, is accessed in both `put` and `take`. Therefore, we use `putLock` both to protect `w` and to serve as the lock on which a taking thread blocks when the queue is empty. Code is shown in Fig. 5.40. Notice that `putLock.wait()` in `get` will release only the lock

```
pubic class SharedQueue4
{
   class Node
   { Object task;
     Node next;

     Node(Object task)
     {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;
   private int w;
   private Object putLock = new Object();
   private Object takeLock = new Object();

   public void put(Object task)
   { synchronized(putLock)
     { assert task != null: "Cannot insert null task";
       Node p = new Node(task);
       last.next = p;
       last = p;
       if(w>0){putLock.notify();}
     }
   }

   public Object take()
   { Object task = null;
     synchronized(takeLock)
     { //returns first task in queue, waits if queue is empty
       while (isEmpty())
      { try{synchronized(putLock){w++; putLock.wait();w--;} }
          catch(InterruptedException error){assert false;}}
       { Node first = head.next;
         task = first.task;
         first.task = null;
         head = first;
       }
     }
     return task;
   }

   private boolean isEmpty(){return head.next == null;}

}
```

Figure 5.40: Blocking queue with multiple locks to allow concurrent `put` and `take` on a nonempty queue

on `putLock`, so a blocked thread will continue to block other takers from the outer block synchronized on `takeLock`. This is okay for this particular problem. This scheme continues to allow putters and takers to execute concurrently; the only exception being when the queue is empty.

Another issue to note is that this solution has nested synchronized blocks in both `take` and `put`. Nested synchronized blocks should always be examined for potential deadlocks. In this case, there will be no deadlock because `put` only acquires one lock, `putLock`. More generally, we would define a partial order over all the locks and ensure that the locks are always acquired in an order consistent with our partial order. For example, here, we could define `takeLock` < `putLock` and make sure that the synchronized blocks are entered in a way that respects that partial order.

As mentioned earlier, several Java-based implementations of queues are included in Java 2 1.5 in the `java.util.concurrent` package, some based on the simple strategies discussed here and some based on more complex strategies that provide additional flexibility and performance.

Distributed shared queues. A centralized shared queue may cause a hot spot, indicating that performance might be improved by a more distributed implementation. As an example, we will develop a simple package to support fork/join programs using a pool of threads and a distributed task queue in the underlying implementation. The package is a much simplified version of the FJTask package [Lea00b], which in turn uses ideas from [BJK+96]. The idea is to create a fixed pool of threads to execute the tasks that are dynamically created as the program executes. Instead of a single central task queue, we associate a nonblocking queue with each thread. When a thread generates a new task, it is placed in its own queue. When a thread is able to execute a new task, it first tries to obtain a task from its own queue. If its own queue is empty, it randomly chooses another thread and attempts to steal a task from that thread's queue and continues checking the other queues until a task is found. (In [BJK+96], this is called *random work stealing.*)

A thread terminates when it receives a poison-pill task. For the fork/join programs we have in mind, this approach has been shown to work well when threads remove tasks from their own queue in LIFO (last in, first out) order and from other queues in FIFO (first in, first out) order. Therefore, we will add to the ADT an operation that removes the last element, to be used by threads to remove tasks from their own queues. The implementation can then be similar to Fig. 5.40, but with an additional method `takeLast` for the added operation. The result is shown in Fig. 5.41.

The remainder of the package comprises three classes.

- `Task` is an abstract class. Applications extend it and override its `run` method to indicate the functionality of a task in the computation. Methods offered by the class include `fork` and `join`.

- `TaskRunner` extends `Thread` and provides the functionality of the threads in the thread pool. Each instance contains a shared task queue. The task-stealing code is in this class.


```
public class SharedQueue5
{
   class Node
   { Object task;
      Node next;
      Node prev;

      Node(Object task, Node prev)
      {this.task = task; next = null; this.prev = prev;}
   }

   private Node head = new Node(null, null);
   private Node last = head;

   public synchronized void put(Object task)
   { assert task != null: "Cannot insert null task";
      Node p = new Node(task, last);
      last.next = p;
      last = p;
   }

   public synchronized Object take()
   { //returns first task in queue or null if queue is empty
      Object task = null;
      if (!isEmpty())
      { Node first = head.next;
         task = first.task;
         first.task = null;
         head = first;
      }
      return task;
   }

   public synchronized Object takeLast()
   { //returns last task in queue or null if queue is empty
      Object task = null;
      if (!isEmpty())
      { task = last.task; last = last.prev; last.next = null;}
      return task;
   }

   private boolean isEmpty(){return head.next == null;}
}
```

Figure 5.41: Nonblocking shared queue with `takeLast` operation

- `TaskRunnerGroup` manages the `TaskRunners`. It contains methods to initialize and shut down the thread pool. It also has a method `executeAndWait` that starts a task running and waits for its completion. This method is used to get the computation started. (It is needed because the `fork` method in class `Task` can only be invoked from within a `Task`. We describe the reason for this restriction later.)

```
public abstract class Task implements Runnable
{
  //done indicates whether the task is finished
   private volatile boolean done;
   public final void setDone(){done = true;}
   public boolean isDone(){return done;}

  //returns the currently executing TaskRunner thread
   public static TaskRunner getTaskRunner()
   { return (TaskRunner)Thread.currentThread(); }

  //push this task on the local queue of current thread
   public void fork()
   { getTaskRunner().put(this);
   }

  //wait until this task is done
   public void join()
   { getTaskRunner().taskJoin(this);
   }

  //execute the run method of this task
   public void invoke()
   { if (!isDone()){run(); setDone(); }
   }
}
```

Figure 5.42: Abstract base class for tasks

We will now discuss these classes in more detail. `Task` is shown in Fig. 5.42. The only state associated with the abstract class is `done`, which is marked `volatile` to ensure that any thread that tries to access it will obtain a fresh value.

The `TaskRunner` class is shown in Fig. 5.43, Fig. 5.44, and Fig. 5.45. The thread, as specified in the `run` method, loops until the poison task is encountered. First it tries to obtain a task from the back of its local queue. If the local queue is empty, it attempts to steal a task from the front of a queue belonging to another thread.

The code for the `TaskRunnerGroup` class is shown in Fig. 5.46. The constructor for `TaskRunnerGroup` initializes the thread pool, given the number of threads as a parameter. Typically, this value would be chosen to match the number of processors in the system. The `executeAndWait` method starts a task by placing it in the task queue of thread 0.

One use for this method is get a computation started. Something like this is needed because we can't just fork a new `Task` from a main or other non-`TaskRunner` thread—this is what was meant by the earlier remark that the `fork` and `join` methods of `Task` can only be invoked from within another `Task`. This is because these methods require interaction with the `TaskRunner` thread executing


```
import java.util.*;

class TaskRunner extends Thread
{

   private final TaskRunnerGroup g; //managing group
   private final Random chooseToStealFrom; //random number generator
   private final Task poison; //poison task
   protected volatile boolean active; //state of thread
   final int id; //index of task in the TaskRunnerGroup

   private final SharedQueue5 q; //Nonblocking shared queue

   //operations relayed to queue
   public void put(Task t){q.put(t);}
   public Task take(){return (Task)q.take();}
   public Task takeLast(){return (Task)q.takeLast();}

   //constructor
   TaskRunner(TaskRunnerGroup g, int id, Task poison)
   { this.g = g;
      this.id = id;
      this.poison = poison;
      chooseToStealFrom = new Random(System.identityHashCode(this));
      setDaemon(true);
      q = new SharedQueue5();
   }

   protected final TaskRunnerGroup getTaskRunnerGroup(){return g;}
   protected final int getID(){return id;}
/* continued in next figure */
```

Figure 5.43: Class defining behavior of threads in the thread pool (continued in Fig. 5.44 and Fig. 5.45)

the task (for example, `fork` involves adding the task to the thread's task queue); we find the appropriate `TaskRunner` using `Thread.getCurrentThread`, thus `fork` and `join` must be invoked only in code being executed by a thread that is a `TaskRunner`.

We normally also want the program that creates the initial task to wait until it completes before going on. To accomplish this and also meet the restriction on when `fork` can be invoked on a `task`, we create a "wrapper" task whose function is to start the initial task, wait for it to complete, and then notify the main thread (the one that called `executeAndWait`). We then add this wrapper task to thread 0's task queue, making it eligible to be executed, and wait for it to notify us (with `notifyAll`) that it has completed.

All of this may be clearer from the usage of `fork`, `join`, and `executeAndWait` in the Fibonacci example in the Examples section.

```
/* continued from previous figure */
   //Attempts to steal a task from another thread. First chooses a
   //random victim, then continues with other threads until either
   //a task has been found or all have been checked. If a task
   //is found, it is invoked. The parameter waitingFor is a task
   //on which this thread is waiting for a join. If steal is not
   //called as part of a join, use waitingFor = null.
   void steal(final Task waitingFor)
   { Task task = null;

      TaskRunner[] runners = g.getRunners();
      int victim = chooseToStealFrom.nextInt(runners.length);
          for (int i = 0; i != runners.length; ++i)
        { TaskRunner tr = runners[victim];
          if (waitingFor != null && waitingFor.isDone()){break;}
        else
        { if (tr != null && tr != this)
          task = (Task)tr.q.take();
          if(task != null) {break;}
          yield();
          victim = (victim + 1)%runners.length;

        }
      } //have either found a task or have checked all other queues

       //if have a task, invoke it
       if(task != null && ! task.isDone())
       { task.invoke(); }
   }
/* continued in next figure
```

Figure 5.44: Class defining behavior of threads in the thread pool (continued from Fig. 5.43 and continued in Fig. 5.45)

Examples

Computing Fibonacci numbers. We show in Fig. 5.47 and Fig. 5.48 code that uses our distributed queue package.[8] Recall that

$$Fib(0) = 0 \tag{5.7}$$

$$Fib(1) = 1 \tag{5.8}$$

$$Fib(n+2) = Fib(n) + Fib(n+1) \tag{5.9}$$

This is a classic divide-and-conquer algorithm. To use our task package, we define a class `Fib` that extends `Task`. Each `Fib` task contains a member `number` that

[8]This code is essentially the same as the class to compute Fibonacci numbers that is provided as a demo with the FJTask package, except for the slight modification necessary to use the classes described previously.


```
/* continued from previous figure */
  //Main loop of thread. First attempts to find a task on local
  //queue and execute it. If not found, then tries to steal a task
  //from another thread. Performance may be improved by modifying
  //this method to back off using sleep or lowered priorities if the
  //thread repeatedly iterates without finding a task. The run
  //method, and thus the thread, terminates when it retrieves the
  //poison task from the task queue.
   public void run()
   { Task task = null;
      try
      { while (!poison.equals(task))
        { task = (Task)q.takeLast();
          if (task != null) { if (!task.isDone()){task.invoke();}}
          else { steal(null); }
        }
       } finally { active = false; }
    }

   //Looks for another task to run and continues when Task w is done.
    protected final void taskJoin(final Task w)
    { while(!w.isDone())
       { Task task = (Task)q.takeLast();
          if (task != null) { if (!task.isDone()){ task.invoke();}}
          else { steal(w);}
       }
    }
}
```

Figure 5.45: Class defining behavior of threads in the thread pool (continued from Fig. 5.43 and Fig. 5.44)

initially contains the number for which the Fibonacci number should be computed and later is replaced by the result. The `getAnswer` method returns the result after it has been computed. Because this variable will be accessed by multiple threads, it is declared `volatile`.

The `run` method defines the behavior of each task. Recursive parallel decomposition is done by creating a new `Fib` object for each subtask, invoking the `fork` method on each subtask to start their computation, calling the `join` method for each subtask to wait for the subtasks to complete, and then computing the sum of their results.

The `main` method drives the computation. It first reads `proc` (the number of threads to create), `num` (the value for which the Fibonacci number should be computed), and optionally the `sequentialThreshold`. The value of this last, optional parameter (the default is 0) is used to decide when the problem is too small to bother with a parallel decomposition and should therefore use a sequential algorithm. After these parameters have been obtained, the `main` method creates a `TaskRunnerGroup` with the indicated number of threads, and then creates a `Fib` object, initialized with `num`. The computation is initiated by passing the `Fib` object to the `TaskRunnerGroup`'s `invokeAndWait` method. When this returns, the

```
class TaskRunnerGroup
{ protected final TaskRunner[] threads;
  protected final int groupSize;
  protected final Task poison;

  public TaskRunnerGroup(int groupSize)
  { this.groupSize = groupSize;
    threads = new TaskRunner[groupSize];
    poison = new Task(){public void run(){assert false;}};
    poison.setDone();
    for (int i = 0; i!= groupSize; i++)
      {threads[i] = new TaskRunner(this,i,poison);}
    for(int i=0; i!= groupSize; i++){ threads[i].start(); }
  }

  //start executing task t and wait for its completion.
  //The wrapper task is used in order to start t from within
  //a Task (thus allowing fork and join to be used)
  public void executeAndWait(final Task t)
  { final TaskRunnerGroup thisGroup = this;
    Task wrapper = new Task()
     { public void run()
       { t.fork();
         t.join();
         setDone();
         synchronized(thisGroup)
           { thisGroup.notifyAll();} //notify waiting thread
       }
     };
    //add wrapped task to queue of thread[0]
    threads[0].put(wrapper);
    //wait for notification that t has finished.
    synchronized(thisGroup)
    { try{thisGroup.wait();}
      catch(InterruptedException e){return;}
    }
  }

  //cause all threads to terminate. The programmer is responsible
  //for ensuring that the computation is complete.
  public void cancel()
  { for(int i=0; i!= groupSize; i++)
    { threads[i].put(poison); }
  }

  public TaskRunner[] getRunners(){return threads;}
}
```

Figure 5.46: The `TaskRunnerGroup` class. This class initializes and manages the threads in the thread pool.

computation is finished. The thread pool is shut down with the `TaskRunnerGroup`'s `cancel` method. Finally, the result is retrieved from the `Fib` object and displayed.

Related Patterns

The *Shared Queue* pattern is an instance of the *Shared Data* pattern. It is often used to represent the task queues in algorithms that use the *Master/Worker* pattern.


```
public class Fib extends Task
{
  volatile int number; // number holds value to compute initially,
                       //after computation is replaced by answer
  Fib(int n) { number = n; } //task constructor, initializes number

  //behavior of task
  public void run() {
    int n = number;

    // Handle base cases:
    if (n <= 1) { // Do nothing: fib(0) = 0; fib(1) = 1 }
    // Use sequential code for small problems:
    else if (n <= sequentialThreshold) {
      number = seqFib(n);
    }
    // Otherwise use recursive parallel decomposition:
    else {
      // Construct subtasks:
      Fib f1 = new Fib(n - 1);
      Fib f2 = new Fib(n - 2);

      // Run them in parallel:
      f1.fork();f2.fork();
      // Await completion;
      f1.join();f2.join();

      // Combine results:
      number = f1.number + f2.number;
      // (We know numbers are ready, so directly access them.)
    }
  }

  // Sequential version for arguments less than threshold
  static int seqFib(int n) {
    if (n <= 1) return n;
    else return seqFib(n-1) + seqFib(n-2);
  }

  //method to retrieve answer after checking to make sure
  //computation has finished, note that done and isDone are
  //inherited from the Task class. done is set by the executing
  //(TaskRunner) thread when the run method is finished.
  int getAnswer() {
    if (!isDone()) throw new Error("Not yet computed");
    return number;
  }
/* continued in next figure */
```

Figure 5.47: Program to compute Fibonacci numbers (continued in Fig. 5.48)

It can also be used to support thread-pool-based implementations of the *Fork/Join* pattern.

Note that when the tasks in a task queue map onto a consecutive sequence of integers, a monotonic shared counter, which would be much more efficient, can be used in place of a queue.

```
/* continued from previous figure */
  //Performance-tuning constant, sequential algorithm is used to
  //find Fibonacci numbers for values <= this threshold
  static int sequentialThreshold = 0;

  public static void main(String[] args) {
      int procs; //number of threads
      int num; //Fibonacci number to compute
      try {
        //read parameters from command line
        procs = Integer.parseInt(args[0]);
        num = Integer.parseInt(args[1]);
        if (args.length > 2)
         sequentialThreshold = Integer.parseInt(args[2]);
      }
      catch (Exception e) {
        System.out.println("Usage: java Fib <threads> <number> "+
            "[<sequentialThreshold>]");
        return;
      }

      //initialize thread pool
      TaskRunnerGroup g = new TaskRunnerGroup(procs);

      //create first task
      Fib f = new Fib(num);

      //execute it
      g.executeAndWait(f);

      //computation has finished, shutdown thread pool
      g.cancel();

      //show result
      long result;
      {result = f.getAnswer();}
      System.out.println("Fib: Size: " + num + " Answer: " + result);
  }

}
```

Figure 5.48: Program to compute Fibonacci numbers (continued from Fig. 5.47)

5.10 THE *DISTRIBUTED ARRAY* PATTERN

Problem

Arrays often need to be partitioned between multiple UEs. How can we do this so the resulting program that is both readable and efficient?

Context

Large arrays are fundamental data structures in scientific computing problems. Differential equations are at the core of many technical computing problems, and solving these equations requires the use of large arrays that arise naturally when a continuous domain is replaced by a collection of values at discrete points. Large arrays also arise in signal processing, statistical analysis, global optimization, and a host of other problems. Hence, it should come as no surprise that dealing effectively with large arrays is an important problem.

If parallel computers were built with a single address space that was large enough to hold the full array yet provided equal-time access from any PE to any array element, we would not need to invest much time in how these arrays are handled. But processors are much faster than large memory subsystems, and networks connecting nodes are much slower than memory buses. The end result is usually a system in which access times vary substantially depending on which PE is accessing which array element.

The challenge is to organize the arrays so that the elements needed by each UE are nearby at the right time in the computation. In other words, the arrays must be distributed about the computer so that the array distribution matches the flow of the computation.

This pattern is important for any parallel algorithm involving large arrays in a parallel algorithm. It is particularly important when the algorithm uses the *Geometric Decomposition* pattern for its algorithm structure and the *SPMD* pattern for its program structure. Although this pattern is in some respects specific to distributed-memory environments in which global data structures must be somehow distributed among the ensemble of PEs, some of the ideas of this pattern apply if the single address space is implemented on a NUMA platform, in which all PEs have access to all memory locations, but access time varies. For such platforms, it is not necessary to explicitly decompose and distribute arrays, but it is still important to manage the memory hierarchy so that array elements stay close[9] to the PEs that need them. Because of this, on NUMA machines, MPI programs can sometimes outperform similar algorithms implemented using a native multithreaded API. Further, the ideas of this pattern can be used with a multithreaded API to keep memory pages close to the processors that will work with them. For example, if the target system uses a first touch page-management scheme, efficiency is improved if every array element is initialized by the PE that will be working with it. This strategy, however, breaks down if arrays need to be remapped in the course of the computation.

Forces

- **Load balance.** Because a parallel computation is not finished until all UEs complete their work, the computational load among the UEs must be distributed so each UE takes nearly the same time to compute.

[9] NUMA computers are usually built from hardware modules that bundle together processors and a subset of the total system memory. Within one of these hardware modules, the processors and memory are "close" together and processors can access this "close" memory in much less time than for remote memory.

- **Effective memory management.** Modern microprocessors are much faster than the computer's memory. To address this problem, high-performance computer systems include complex memory hierarchies. Good performance depends on making good use of this memory hierarchy, and this is done by ensuring that the memory references implied by a series of calculations are close to the processor making the calculation (that is, data reuse from the caches is high and needed pages stay accessible to the processor).

- **Clarity of abstraction.** Programs involving distributed arrays are easier to write, debug, and maintain if it is clear how the arrays are divided among UEs and mapped to local arrays.

Solution

Overview. The solution is simple to state at a high level; it is the details that make it complicated. The basic approach is to partition the global array into blocks and then map those blocks onto the UEs. This mapping onto UEs should be done so that, as the computation unfolds, each UE has an equal amount of work to carry out (that is, the load must be well balanced). Unless all UEs share a single address space, each UE's blocks will be stored in an array that is local to a single UE. Thus, the code will access elements of the distributed array using indices into a local array. The mathematical description of the problem and solution, however, is based on indices into the global array. Thus, it must be clear how to move back and forth between two views of the array, one in which each element is referenced by global indices and one in which it is referenced by a combination of local indices and UE identifier. Making these translations clear within the text of the program is the challenge of using this pattern effectively.

Array distributions. Over the years, a small number of array distributions have become standard.

- **One-dimensional (1D) block.** The array is decomposed in one dimension only and distributed one block per UE. For a 2D matrix, for example, this corresponds to assigning a single block of contiguous rows or columns to each UE. This distribution is sometimes called a *column block* or *row block* distribution depending on which single dimension is distributed among the UEs. The UEs are conceptually organized as a 1D array.

- **Two-dimensional (2D) block.** As in the 1D block case, one block is assigned to each UE, but now the block is a rectangular subblock of the original global array. This mapping views the collection of UEs as a 2D array.

- **Block-cyclic.** The array is decomposed into blocks (using a 1D or 2D partition) such that there are more blocks than UEs. These blocks are then assigned round-robin to UEs, analogous to the way a deck of cards is dealt out. The UEs may be viewed as either a 1D or 2D array.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

Figure 5.49: Original square matrix A

Next, we explore these distributions in more detail. For illustration, we use a square matrix A of order 8, as shown in Fig. 5.49.[10]

1D block. Fig. 5.50 shows a column block distribution of A onto a linear array of four UEs. The matrix is decomposed along the column index only; the number of columns in each block, MB (2 here), is the matrix order divided by the number of UEs. Matrix element (i, j) is assigned to $UE(j \backslash MB)$.[11]

Mapping to UEs. More generally, we could have an $N \times M$ matrix where the number of UEs, P, need not divide the number of columns evenly. In this case, MB is the maximum number of columns mapped to a UE, and all UEs except $UE(P-1)$ contain MB blocks. Then, $MB = \lceil M/P \rceil$, and elements of column j are mapped to $UE(\lfloor j/MB \rfloor)$.[12] (This reduces to the formula given earlier for the

[10] In this and the other figures in this pattern, we will use the following notational conventions: A matrix element will be represented as a lowercase letter with subscripts representing indices; for example, $a_{1,2}$ is the element in row 1 and column 2 of matrix A. A submatrix will be represented as an uppercase letter with subscripts representing indices; for example, $A_{0,0}$ is a submatrix containing the top-left corner of A. When we talk about assigning parts of A to UEs, we will reference different UEs using UE and an index or indices in parentheses; for example, if we are regarding UEs as forming a 1D array, $UE(0)$ is the conceptually leftmost UE, while if we are regarding UEs as forming a 2D array, $UE(0,0)$ is the conceptually top-left UE. Indices are all assumed to be zero-based (that is, the smallest index is 0).

[11] We will use the notation "\" for integer division, and "/" for normal division. Thus $a \backslash b = \lfloor a/b \rfloor$. Also, $\lfloor x \rfloor$ (floor) is the largest integer at most x, and $\lceil x \rceil$ (ceiling) is the smallest integer at least x. For example, $\lfloor 4/3 \rfloor = 1$, and $\lfloor 4/2 \rfloor = 2$.

[12] Notice that this is not the only possible way to distribute columns among UEs when the number of UEs does not evenly divide the number of columns. Another approach, more complex to define but producing a more balanced distribution in some cases, is to first define the minimum number of columns per UE as $\lfloor M/P \rfloor$, and then increase this number by one for the first (M mod P) UEs. For example, for $M = 10$ and $P = 4$, $UE(0)$ and $UE(1)$ would have three columns each and $UE(2)$ and $UE(3)$ would have two columns each.

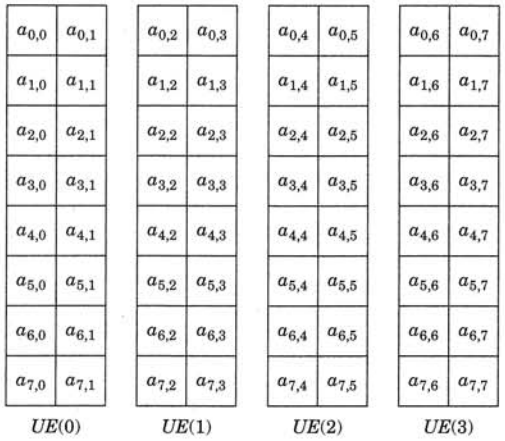

Figure 5.50: 1D distribution of *A* onto four UEs

example, because in the special case where P evenly divides M, $\lceil M/P \rceil = M/P$ and $\lfloor j/MB \rfloor = j/MB$.) Analogous formulas apply for row distributions.

Mapping to local indices. In addition to mapping the columns to UEs, we also need to map the global indices to local indices. In this case, matrix element (i, j) maps to local element $(i, j \bmod MB)$. Given local indices (x, y) and $UE(w)$, we can recover the global indices $(x, wMB + y)$. Again, analogous formulas apply for row distributions.

2D block. Fig. 5.51 shows a 2D block distribution of A onto a two-by-two array of UEs. Here, A is being decomposed along two dimensions, so for each subblock, the number of columns is the matrix order divided by the number of columns of UEs, and the number of rows is the matrix order divided by the number of rows of UEs. Matrix element (i, j) is assigned to $UE(i \backslash 2, j \backslash 2)$.

Mapping to UEs. More generally, we map an $N \times M$ matrix to a $P_R \times P_C$ matrix of UEs. The maximum size of a subblock is $NB \times MB$, where $NB = \lceil N/P_R \rceil$ and $MB = \lceil M/P_C \rceil$. Then, element (i, j) in the global matrix is stored in $UE(\lfloor i/NB \rfloor, \lfloor j/MB \rfloor$.

Mapping to local indices. Global indices (i, j) map to local indices $(i \bmod NB, j \bmod MB)$. Given local indices (x, y) on $UE(z, w)$ the corresponding global indices are $(zNB + x, wMB + y)$.

Block-cyclic. The main idea behind the block-cyclic distribution is to create more blocks than UEs and allocate them in a cyclic manner, similar to dealing

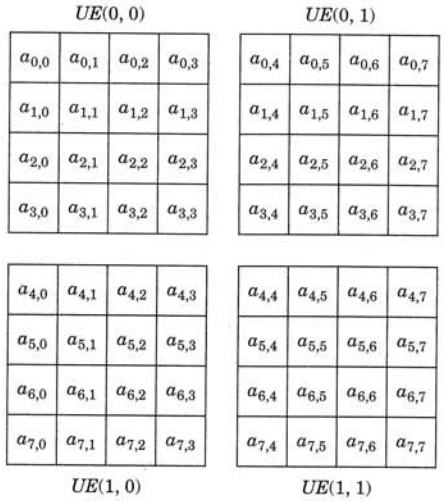

Figure 5.51: 2D distribution of *A* onto four UEs

out a deck of cards. Fig. 5.52 shows a 1D block-cyclic distribution of A onto a linear array of four UEs, illustrating how columns are assigned to UEs in a round-robin fashion. Here, matrix element (i, j) is assigned to $UE(j \bmod 4)$ (where 4 is the number of UEs).

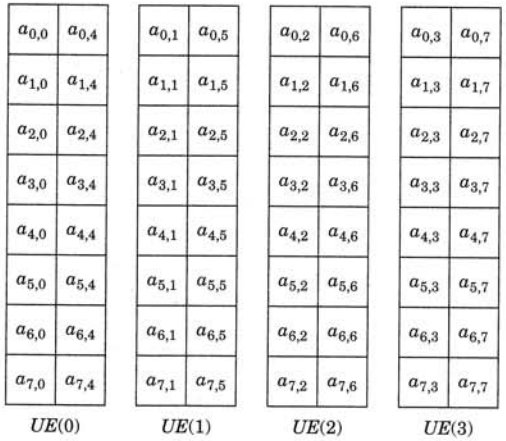

Figure 5.52: 1D block-cyclic distribution of *A* onto four UEs

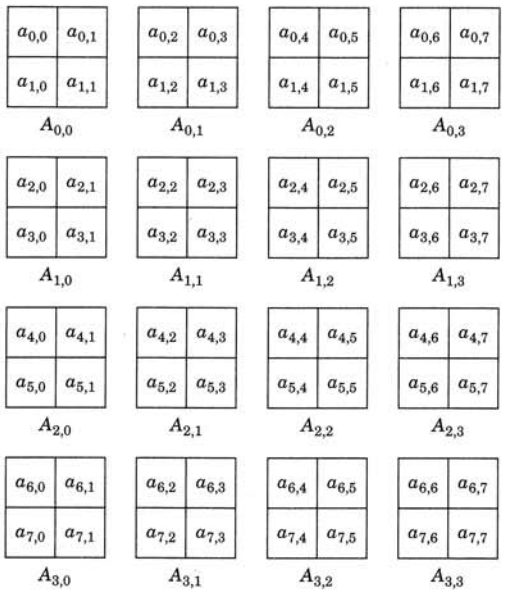

Figure 5.53: 2D block-cyclic distribution of *A* onto four UEs, part 1: Decomposing *A*

Fig. 5.53 and Fig. 5.54 show a 2D block-cyclic distribution of A onto a two-by-two array of UEs: Fig. 5.53 illustrates how A is decomposed into two-by-two submatrices. (We could have chosen a different decomposition, for example one-by-one submatrices, but two-by-two illustrates how this distribution can have both block and cyclic characteristics.) Fig. 5.54 then shows how these submatrices are assigned to UEs. Matrix element (i, j) is assigned to $UE(i \bmod 2, j \bmod 2)$.

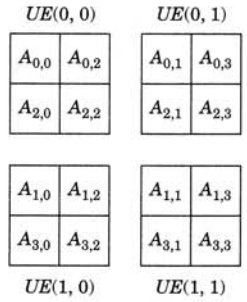

Figure 5.54: 2D block-cyclic distribution of *A* onto four UEs, part 2: Assigning submatrices to UEs

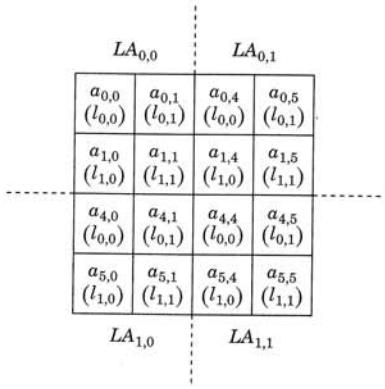

Figure 5.55: 2D block-cyclic distribution of *A* onto four UEs: Local view of elements of *A* assigned to *UE*(0,0). $LA_{l,m}$ is the block with block indices (l, m). Each element is labeled both with its original global indices ($a_{i,j}$) and its indices within block $LA_{l,m}$ ($l_{x,y}$).

Mapping to UEs. In the general case, we have an $N \times M$ matrix to be mapped onto a $P_R \times P_C$ array of UEs. We choose block size $NB \times MB$. Element (i, j) in the global matrix will be mapped to $UE(z, w)$, where $z = \lfloor i/NB \rfloor \bmod P_R$ and $w = \lfloor j/MB \rfloor \bmod P_C$.

Mapping to local indices. Because multiple blocks are mapped to the same UE, we can view the local indexing blockwise or elementwise.

In the blockwise view, each element on a UE is indexed locally by block indices (l, m) and indices (x, y) into the block. To restate this: In this scheme, the global matrix element (i, j) will be found on the UE within the local (l, m) block at the position (x, y) where $(l, m) = (\lfloor i/(P_R\, NB) \rfloor, \lfloor j/(P_C\, MB) \rfloor)$ and $(x, y) = (i \bmod NB, j \bmod MB)$. Fig. 5.55 illustrates this for $UE(0, 0)$.

For example, consider global matrix element $a_{5,1}$. Because $P_R = P_C = NB = MB = 2$, this element will map to $UE(0, 0)$. There are four two-by-two blocks on this UE. From the figure, we see that this element appears in the block on the bottom left, or block $LA_{1,0}$ and indeed, from the formulas, we obtain $(l, m) = (\lfloor 5/(2 \times 2) \rfloor, \lfloor 1/(2 \times 2) \rfloor) = (1, 0)$. Finally, we need the local indices within the block. In this case, the indices within block are $(x, y) = (5 \bmod 2, 1 \bmod 2) = (1, 1)$.

In the elementwise view (which requires that all the blocks for each UE form a contiguous matrix), global indices (i, j) are mapped elementwise to local indices $(lNB + x, mMB + y)$, where l and m are defined as before. Fig. 5.56 illustrates this for $UE(0, 0)$.

Again, looking at global matrix element $a_{5,1}$, we see that viewing the data as a single matrix, the element is found at local indices $(1 \times 2 + 1, 0 \times 2 + 1) = (3, 1)$. Local indices (x, y) in block (l, m) on $UE(z, w)$ correspond to global indices $((lP_R + z)NB + x, (mP_C + w)MB + y)$.

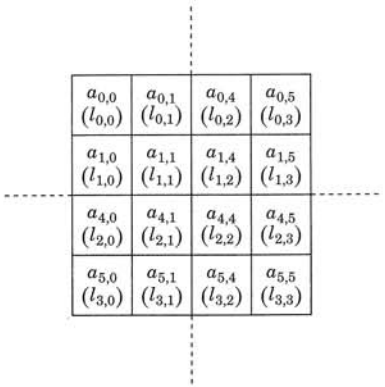

Figure 5.56: 2D block-cyclic distribution of *A* onto four UEs: Local view of elements of *A* assigned to *UE*(0,0). Each element is labeled both with its original global indices $a_{i,j}$ and its local indices $l_{x',y'}$. Local indices are with respect to the contiguous matrix used to store all blocks assigned to this UE.

Choosing a distribution. To select which distribution to use for a problem, consider how the computational load on the UEs changes as the computation proceeds. For example, in many single-channel signal-processing problems, the same set of operations is performed on each column of an array. The work does not vary as the computation proceeds, so a column block decomposition will produce both clarity of code and good load balance. If instead the amount of work varies by column, with higher-numbered columns requiring more work, a column block decomposition would lead to poor load balance, with the UEs processing lower-numbered columns finishing ahead of the UEs processing higher-numbered columns. In this case, a cyclic distribution would produce better load balance, because each UE is assigned a mix of low-numbered and high-numbered columns.

This same approach applies to higher dimensions as well. ScaLAPACK [Sca, BCC+97], the leading package of dense linear algebra software for distributed-memory computers, requires a 2D block-cyclic distribution. To see why this choice was made, consider Gaussian elimination, one of the more commonly used of the ScaLAPACK routines. In this algorithm, also known as LU decomposition, a dense square matrix is transformed into a pair of triangular matrices, an upper matrix U and a lower matrix L. At a high level, the algorithm proceeds from the upper-left corner and works its way down the diagonal of the global matrix, eliminating elements below the diagonal and transforming the remaining blocks to the right as needed. A block distribution would result in idle UEs as the processing marches down the diagonal. But with a 2D block-cyclic distribution such as the one shown in Fig. 5.54, each UE contains elements used both early and late in the algorithm, resulting in excellent load balance.

Mapping indices. The examples in the preceding section illustrate how each element of the original (global) array is mapped to a UE and how each element

in the global array, after distribution, is identified by both a set of global indices and a combination of UE identifier and local information. The original problem is typically stated in terms of global indices, but computation within each UE must be in terms of local indices. Applying this pattern effectively requires that the relationship between global indices and the combination of UE and local indices be as transparent as possible. In a quest for program efficiency, it is altogether too easy to bury these index mappings in the code in a way that makes the program painfully difficult to debug. A better approach is to use macros and inline functions to capture the index mappings; a human reader of the program then only needs to master the macro or function once. Such macros or functions also contribute to clarity of abstraction. The Examples section illustrates this strategy.

Aligning computation with locality. One of the cardinal rules of performance-oriented computing is to maximize reuse of data close to a UE. That is, the loops that update local data should be organized in a way that gets as much use as possible out of each memory reference. This objective can also influence the choice of array distribution.

For example, in linear algebra computations, it is possible to organize computations on a matrix into smaller computations over submatrices. If these submatrices fit into cache, dramatic performance gains can result. Similar effects apply to other levels of the memory hierarchy: minimizing misses in the translation lookaside buffer (TLB), page faults, and so on. A detailed discussion of this topic goes well beyond the scope of this book. An introduction can be found in [PH98].

Examples

Transposing a matrix stored as column blocks. As an example of organizing matrix computations into smaller computations over submatrices, consider transposing a square matrix distributed with a column block distribution. For simplicity, we will assume that the number of UEs evenly divides the number of columns, so that all blocks are the same size. Our strategy for transposing the matrix will be based on logically decomposing the matrix into square submatrices, as shown in Fig. 5.57. Each of the labeled blocks in the figure represents a square submatrix; labels show how the blocks of the transpose relate to the blocks of the

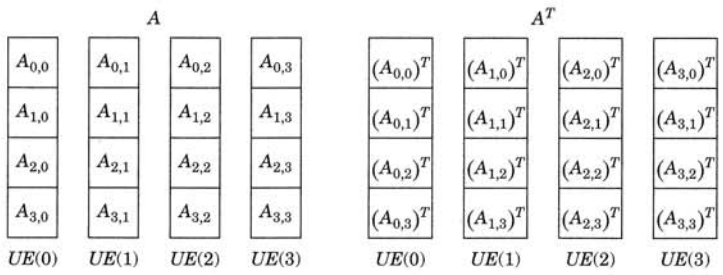

Figure 5.57: Matrix *A* and its transpose, in terms of submatrices, distributed among four UEs

original matrix. (For example, the block labeled $(A_{0,1})^T$ in the transpose is the transpose of the block labeled $A_{0,1}$ in the original matrix.) The algorithm proceeds in phases; the number of phases is the number of submatrices per UE (which is also the number of UEs). In the first phase, we transpose the submatrices on the diagonal of A, with each UE transposing one submatrix and no communication required. In successive phases, we transpose the submatrices one below the diagonal, then two below the diagonal, and so forth, wrapping around to the top of the matrix as necessary. In each of these phases, each UE must transpose one of its submatrices, send it to another UE, and receive a submatrix. For example, in the second phase, the UE labeled $UE(1)$ must compute $(A_{2,1})^T$, send it to $UE(2)$, and receive $(A_{0,1})^T$ from $UE(0)$. Figs. 5.58 and 5.59 show code to transpose such a matrix. This code represents a function that will transpose a square column-blocked array. We assume the blocks are distributed contiguously with one column block per UE. This function is intended as part of a larger program, so we assume the array has already been distributed prior to calling this function.

The program represents each local column block (one for A and one for the transposed result) as a 1D array. These arrays in turn consist of `Num_procs` submatrices each, each of size `block_size = Block_order * Block_order`, where

```
/*********************************************************************

NAME: trans_isend_ircv

PURPOSE: This function uses MPI Isend and Irecv to transpose
        a column-block distributed matrix.

*********************************************************************/

#include "mpi.h"
#include <stdio.h>

/*********************************************************************
** This function transposes a local block of a matrix. We don't
** display the text of this function as it is not relevant to the
** point of this example.
*********************************************************************/
void transpose(
   double* A, int Acols, /* input matrix */
   double* B, int Bcols, /* transposed mat */
   int sub_rows, int sub_cols); /* size of slice to transpose */

/*********************************************************************
** Define macros to compute process source and destinations and
** local indices
*********************************************************************/
#define TO(ID, PHASE, NPROC) ((ID + PHASE ) % NPROC)
#define FROM(ID, PHASE, NPROC) ((ID + NPROC - PHASE) % NPROC)
#define BLOCK(BUFF, ID) (BUFF + (ID * block_size))
/* continued in next figure */
```

Figure 5.58: Code to transpose a matrix (continued in Fig. 5.59)


```
/* continued from previous figure */

void trans_isnd_ircv(double *buff, double *trans, int Block_order,
                double *work, int my_ID, int num_procs)
{
   int iphase;
   int block_size;
   int send_to, recv_from;
   double *bblock; /* pointer to current location in buff */
   double *tblock; /* pointer to current location in trans */
   MPI_Status status;
   MPI_Request send_req, recv_req;

   block_size = Block_order * Block_order;

/*********************************************************************
** Do the transpose in num_procs phases.
**
** In the first phase, do the diagonal block. Then move out
** from the diagonal copying the local matrix into a communication
** buffer (while doing the local transpose) and send to process
** (diag+phase)%num_procs.
*********************************************************************/
   bblock = BLOCK(buff, my_ID);
   tblock = BLOCK(trans, my_ID);

   transpose(bblock, Block_order, tblock, Block_order,
            Block_order, Block_order);

   for (iphase=1; iphase<num_procs; iphase++){
      recv_from = FROM(my_ID, iphase, num_procs);
      tblock = BLOCK(trans, recv_from);
      MPI_Irecv (tblock, block_size, MPI_DOUBLE, recv_from,
              iphase, MPI_COMM_WORLD, &recv_req);

      send_to = TO(my_ID, iphase, num_procs);
      bblock = BLOCK(buff, send_to);
      transpose(bblock, Block_order, work, Block_order,
            Block_order, Block_order);
      MPI_Isend (work, block_size, MPI_DOUBLE, send_to,
              iphase, MPI_COMM_WORLD, &send_req);

      MPI_Wait(&recv_req, &status);
      MPI_Wait(&send_req, &status);
   }
}
```

Figure 5.59: Code to transpose a matrix (continued from Fig. 5.58)

`Block_order` is the number of columns per UE. We can therefore find the block indexed `ID` using the `BLOCK` macro:

```
#define BLOCK(BUFF, ID) (BUFF + (ID * block_size))
```

`BUFF` is the start of the 1D array (`buff` for the original array, `trans` for the transpose) and `ID` is the second index of the block. So for example, we find the diagonal

block of both arrays as follows:

```
bblock = BLOCK(buff, my_ID);
tblock = BLOCK(trans, my_ID);
```

In succeeding phases of the algorithm, we must determine two things: (1) the index of the block we should transpose and send and (2) the index of the block we should receive. We do this with the `TO` and `FROM` macros:

```
#define TO(ID, PHASE, NPROC) ((ID + PHASE ) % NPROC)
#define FROM(ID, PHASE, NPROC) ((ID + NPROC - PHASE) % NPROC)
```

The `TO` index shows the progression through the off-diagonal blocks, working down from the diagonal and wrapping back to the top at the bottom of the matrix. At each phase of the algorithm, we compute which UE is to receive the block and then update the local pointer (`bblock`) to the block that will be sent:

```
send_to = TO(my_ID, iphase, num_procs);
bblock = BLOCK(buff, send_to);
```

Likewise, we compute where the next block is coming from and which local index corresponds to that block:

```
recv_from = FROM(my_ID, iphase, num_procs);
tblock = BLOCK(trans, recv_from);
```

This continues until all of the blocks have been transposed.

We use immediate (nonblocking) sends and receives in this example. (These primitives are described in more detail in the MPI appendix, Appendix B.) During each phase, each UE first posts a receive and then performs a transpose on the block it will send. After that transpose is complete, the UE sends the now-transposed block to the UE that should receive it. At the bottom of the loop and before moving to the next phase, functions are called to force the UE to wait until both the sends and receives complete. This approach lets us overlap communication and computation. More importantly (because in this case there isn't much computation to overlap with communication), it prevents deadlock: A more straightforward approach using regular sends and receives would be to first transpose the block to be sent, then send it, and then (wait to) receive a block from another UE. However, if the blocks to be sent are large, a regular send might block because there is insufficient buffer space for the message; in this case, such blocking could produce deadlock. By instead using nonblocking sends and receives and posting the receives first, we avoid this situation.

Known uses. This pattern is used throughout the scientific-computing literature. The well-known ScaLAPACK package [Sca, BCC+97] makes heavy use of the 2D block-cyclic distribution, and the documentation gives a thorough explanation of mapping and indexing issues for this distribution.

Several different array distributions were embedded into the HPF language [HPF97] definition.

Some of the most creative uses of this pattern can be found in quantum chemistry, particularly in the area of post Hartree Fock computations. The Global Arrays or GA package [NHL94, NHL96, NHK+02, Gloa] was created specifically to address distributed-array problems in post Hartree Fock algorithms. A more recent approach is described in [NHL96, LDSH95].

The PLAPACK package [ABE+97, PLA, vdG97] takes a different approach to array distribution. Rather than focusing on how to distribute the arrays, PLAPACK considers how the vectors operated upon by the arrays are organized. From these distributed vectors, the corresponding array distributions are derived. In many problems, these vectors correspond to the physical quantities in the problem domain, so the PLAPACK team refers to this as the physically based distribution.

Related Patterns

The *Distributed Array* pattern is often used together with the *Geometric Decomposition* and *SPMD* patterns.

5.11 OTHER SUPPORTING STRUCTURES

This pattern language (and hence the *Supporting Structures* patterns) is based on common practice among OpenMP, MPI, and Java programmers writing code for both shared-memory and distributed-memory MIMD computers. Parallel application programmers will in most cases find the patterns they need within this pattern language.

There are, however, additional patterns (with their own supporting structures) that have at various times been important in parallel programming. They are only rarely used at this time, but it is still important to be aware of them. They can provide insights into different opportunities for finding and exploiting concurrency. And it is possible that as parallel architectures continue to evolve, the parallel programming techniques suggested by these patterns may become important.

In this section, we will briefly describe some of these additional patterns and their supporting structures: *SIMD, MPMD, Client-Server,* and *Declarative Programming*. We close with a brief discussion of problem-solving environments. These are not patterns, but they help programmers work within a targeted set of problems.

5.11.1 SIMD

A SIMD computer has a single stream of instructions operating on multiple streams of data. These machines were inspired by the belief that programmers would find it too difficult to manage multiple streams of instructions. Many important problems are data parallel; that is, the concurrency can be expressed in terms of concurrent updates across the problem's data domain. Carried to its logical extreme, the SIMD

approach assumes that it is possible to express *all* parallelism in terms of the data. Programs would then have single-thread semantics, making understanding and hence debugging them much easier. The basic idea behind the *SIMD* pattern can be summarized as follows.

- Define a network of virtual PEs to be mapped onto the actual PEs. These virtual PEs are connected according to a well-defined topology. Ideally the topology is (1) well-aligned with the way the PEs in the physical machine are connected and (2) effective for the communication patterns implied by the problem being solved.
- Express the problem in terms of arrays or other regular data structures that can be updated concurrently with a single stream of instructions.
- Associate these arrays with the local memories of the virtual PEs.
- Create a single stream of instructions that operates on slices of the regular data structures. These instructions may have an associated mask so they can be selectively skipped for subsets of array elements. This is critical for handling boundary conditions or other constraints.

When a problem is truly data parallel, this is an effective pattern. The resulting programs are relatively easy to write and debug [DKK90].

Unfortunately, most data problems contain subproblems that are not data parallel. Setting up the core data structures, dealing with boundary conditions, and post-processing after a core data parallel algorithm can all introduce logic that might not be strictly data parallel. Furthermore, this style of programming is tightly coupled to compilers that support data-parallel programming. These compilers have proven difficult to write and result in code that is difficult to optimize because it can be far removed from how a program runs on a particular machine. Thus, this style of parallel programming and the machines built around the SIMD concept have largely disappeared, except for a few special-purpose machines used for signal-processing applications.

The programming environment most closely associated with the *SIMD* pattern is High Performance Fortran (HPF) [HPF97]. HPF is an extension of the array-based constructs in Fortran 90. It was created to support portable parallel programming across SIMD machines, but also to allow the SIMD programming model to be used on MIMD computers. This required explicit control over data placement onto the PEs and the capability to remap the data during a calculation. Its dependence on a strictly data-parallel, SIMD model, however, doomed HPF by making it difficult to use with complex applications. The last large community of HPF users is in Japan [ZJS+02], where they have extended the language to relax the data-parallel constraints [HPF99].

5.11.2 MPMD

The *Multiple Program, Multiple Data (MPMD)* pattern, as the name implies, is used in a parallel algorithm when different programs run on different UEs. The

basic approach is the following.

- Decompose the problem into a set of subproblems, where each subproblem maps onto a subset of UEs. Often each subset of UEs corresponds to the nodes of a different parallel computer.
- Create independent programs solving the appropriate subproblems and tuned to the relevant target UEs.
- Coordinate the programs running on distinct UEs as needed, typically through a message-passing framework.

In many ways, the MPMD approach is not too different from an SPMD program using MPI. In fact, the runtime environments associated with the two most common implementations of MPI, MPICH [MPI] and LAM/MPI [LAM], support simple MPMD programming.

Applications of the *MPMD* pattern typically arise in one of two ways. First, the architecture of the UEs may be so different that a single program cannot be used across the full system. This is the case when using parallel computing across some type of computational grid [Glob,FK03] using multiple classes of high-performance computing architectures. The second (and from a parallel-algorithm point of view more interesting) case occurs when completely different simulation programs are combined into a coupled simulation.

For example, climate emerges from a complex interplay between atmospheric and ocean phenomena. Well-understood programs for modeling the ocean and the atmosphere independently have been developed and highly refined over the years. Although an SPMD program could be created that implements a coupled ocean/atmospheric model directly, a more effective approach is to take the separate, validated ocean and atmospheric programs and couple them through some intermediate layer, thereby producing a new coupled model from well-understood component models.

Although both MPICH and LAM/MPI provide some support for MPMD programming, they do not allow different implementations of MPI to interact, so only MPMD programs using a common MPI implementation are supported. To address a wider range of MPMD problems spanning different architectures and different MPI implementations, a new standard called interoperable MPI (iMPI) was created. The general idea of coordinating UEs through the exchange of messages is common to MPI and iMPI, but the detailed semantics are extended in iMPI to address the unique challenges arising from programs running on widely differing architectures. These multi-architecture issues can add significant communication overhead, so the part of an algorithm dependent on the performance of iMPI must be relatively coarse-grained.

MPMD programs are rare. As increasingly complicated coupled simulations grow in importance, however, use of the *MPMD* pattern will increase. Use of this pattern will also grow as grid technology becomes more robust and more widely deployed.

5.11.3 Client-Server Computing

Client-server architectures are related to MPMD. Traditionally, these systems have comprised two or three tiers where the front end is a graphical user interface executed on a client's computer and a mainframe back end (often with multiple processors) provides access to a database. The middle tier, if present, dispatches requests from the clients to (possibly multiple) back ends. Web servers are a familiar example of a client-server system. More generally, a server might offer a variety of services to clients, an essential aspect of the system being that services have well-defined interfaces. Parallelism can appear at the server (which can service many clients concurrently or can use parallel processing to obtain results more quickly for single requests) and at the client (which can initiate requests at more than one server simultaneously).

Techniques used in client-server systems are especially important in heterogeneous systems. Middleware such as CORBA [COR] provides a standard for service interface specifications, enabling new programs to be put together by composing existing services, even if those services are offered on vastly different hardware platforms and implemented in different programming languages. CORBA also provides facilities to allow services to be located. The Java J2EE (Java 2 Platform, Enterprise Edition) [Javb] also provides significant support for client-server applications. In both of these cases, interoperability was a major design force.

Client-server architectures have traditionally been used in enterprise rather than scientific applications. Grid technology, which is heavily used in scientific computing, borrows from client-server technology, extending it by blurring the distinction between clients and servers. All resources in a grid, whether they are computers, instruments, file systems, or anything else connected to the network, are peers and can serve as clients and servers. The middleware provides standards-based interfaces to tie the resources together into a single system that spans multiple administrative domains.

5.11.4 Concurrent Programming with Declarative Languages

The overwhelming majority of programming is done with imperative languages such as C++, Java, or Fortran. This is particularly the case for traditional applications in science and engineering. The artificial intelligence community and a small subset of academic computer scientists, however, have developed and shown great success with a different class of languages, the *declarative languages*. In these languages, the programmer describes a problem, a problem domain, and the conditions solutions must satisfy. The runtime system associated with the language then uses these to find valid solutions.

Declarative semantics impose a different style of programming that overlaps with the approaches discussed in this pattern language, but has some significant differences. There are two important classes of declarative languages: functional languages and logic programming languages.

Logic programming languages are based on formal rules of logical inference. The most common logic programming language by far is Prolog [SS94], a programming language based on first-order predicate calculus. When Prolog is extended to support expression of concurrency, the result is a concurrent logic programming language. Concurrency is exploited in one of three ways with these Prolog extensions: and-parallelism (execute multiple predicates), or-parallelism (execute multiple guards), or through explicit mapping of predicates linked together through single-assignment variables [CG86].

Concurrent logic programming languages were a hot area of research in the late 1980s and early 1990s. They ultimately failed because most programmers were deeply committed to more traditional imperative languages. Even with the advantages of declarative semantics and the value of logic programming for symbolic reasoning, the learning curve associated with these languages proved prohibitive.

The older and more established class of declarative programming languages is based on functional programming models [Hud89]. LISP is the oldest and best known of the functional languages. In pure functional languages, there are no side effects from a function. Therefore, functions can execute as soon as their input data is available. The resulting algorithms express concurrency in terms of the flow of data through the program leading, thereby resulting in "data-flow" algorithms [Jag96].

The best-known concurrent functional languages are Sisal [FCO90], Concurrent ML [Rep99, Con] (an extension to ML), and Haskell [HPF]. Because mathematical expressions are naturally written down in a functional notation, Sisal was particularly straightforward to work with in science and engineering applications and proved to be highly efficient for parallel programming. However, just as with the logic programming languages, programmers were unwilling to part with their familiar imperative languages, and Sisal essentially died. Concurrent ML and Haskell have not made major inroads into high-performance computing, although both remain popular in the functional programming community.

5.11.5 Problem-Solving Environments

A discussion of supporting structures for parallel algorithms would not be complete without mentioning problem-solving environments (PSE). A PSE is a programming environment specialized to the needs of a particular class of problems. When applied to parallel computing, PSEs also imply a particular algorithm structure as well.

The motivation behind PSEs is to spare the application programmer the low-level details of the parallel system. For example, PETsc (Portable, Extensible, Toolkit for Scientific Computation) [BGMS98] supports a variety of distributed data structures and functions required to use them for solving partial differential equations (typically for problems fitting the *Geometric Decomposition* pattern). The programmer needs to understand the data structures within PETSc, but is spared the need to master the details of how to implement them efficiently and portably. Other important PSEs are PLAPACK [ABE+97] (for dense linear algebra problems) and POOMA [RHC+96] (an object-oriented framework for scientific computing).

PSEs have not been very well accepted. PETSc is probably the only PSE that is heavily used for serious application programming. The problem is that by tying themselves to a narrow class of problems, PSEs restrict their potential audience and have a difficult time reaching a critical mass of users. We believe that over time and as the core patterns behind parallel algorithms become better understood, PSEs will be able to broaden their impact and play a more dominant role in parallel programming.