CHAPTER 3

The *Finding Concurrency* Design Space

3.1 ABOUT THE DESIGN SPACE

The software designer works in a number of domains. The design process starts in the *problem domain* with design elements directly relevant to the problem being solved (for example, fluid flows, decision trees, atoms, etc.). The ultimate aim of the design is software, so at some point, the design elements change into ones relevant to a program (for example, data structures and software modules). We call this the *program domain*. Although it is often tempting to move into the program domain as soon as possible, a designer who moves out of the problem domain too soon may miss valuable design options.

This is particularly relevant in parallel programming. Parallel programs attempt to solve bigger problems in less time by simultaneously solving different parts of the problem on different processing elements. This can only work, however, if the problem contains exploitable concurrency, that is, multiple activities or *tasks* that can execute at the same time. After a problem has been mapped onto the program domain, however, it can be difficult to see opportunities to exploit concurrency.

Hence, programmers should start their design of a parallel solution by analyzing the problem within the problem domain to expose exploitable concurrency. We call the design space in which this analysis is carried out the *Finding Concurrency* design space. The patterns in this design space will help identify and analyze the exploitable concurrency in a problem. After this is done, one or more patterns from the *Algorithm Structure* space can be chosen to help design the appropriate algorithm structure to exploit the identified concurrency.

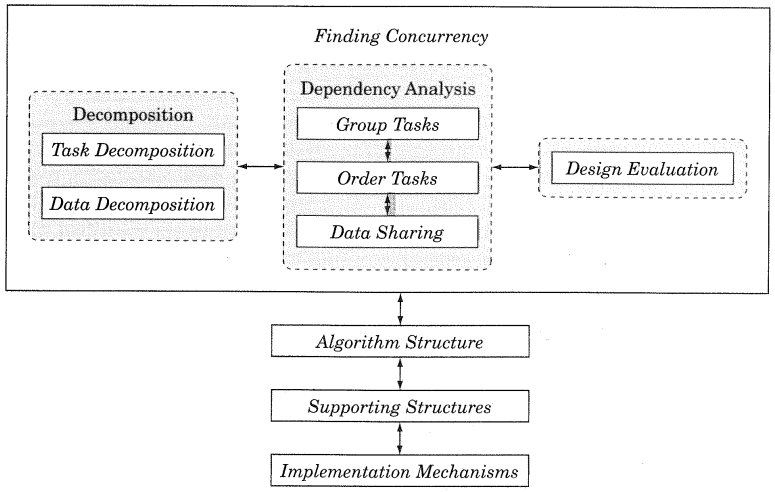

Figure 3.1: Overview of the *Finding Concurrency* design space and its place in the pattern language

An overview of this design space and its place in the pattern language is shown in Fig. 3.1.

Experienced designers working in a familiar domain may see the exploitable concurrency immediately and could move directly to the patterns in the *Algorithm Structure* design space.

3.1.1 Overview

Before starting to work with the patterns in this design space, the algorithm designer must first consider the problem to be solved and make sure the effort to create a parallel program will be justified: Is the problem large enough and the results significant enough to justify expending effort to solve it faster? If so, the next step is to make sure the key features and data elements within the problem are well understood. Finally, the designer needs to understand which parts of the problem are most computationally intensive, because the effort to parallelize the problem should be focused on those parts.

After this analysis is complete, the patterns in the *Finding Concurrency* design space can be used to start designing a parallel algorithm. The patterns in this design space can be organized into three groups.

- **Decomposition Patterns.** The two decomposition patterns, *Task Decomposition* and *Data Decomposition*, are used to decompose the problem into pieces that can execute concurrently.
- **Dependency Analysis Patterns.** This group contains three patterns that help group the tasks and analyze the dependencies among them: *Group Tasks*,

Order Tasks, and *Data Sharing.* Nominally, the patterns are applied in this order. In practice, however, it is often necessary to work back and forth between them, or possibly even revisit the decomposition patterns.

- ***Design Evaluation* Pattern.** The final pattern in this space guides the algorithm designer through an analysis of what has been done so far before moving on to the patterns in the *Algorithm Structure* design space. This pattern is important because it often happens that the best design is not found on the first attempt, and the earlier design flaws are identified, the easier they are to correct. In general, working through the patterns in this space is an iterative process.

3.1.2 Using the Decomposition Patterns

The first step in designing a parallel algorithm is to decompose the problem into elements that can execute concurrently. We can think of this decomposition as occurring in two dimensions.

- The *task-decomposition dimension* views the problem as a stream of instructions that can be broken into sequences called *tasks* that can execute simultaneously. For the computation to be efficient, the operations that make up the task should be largely independent of the operations taking place inside other tasks.
- The *data-decomposition dimension* focuses on the data required by the tasks and how it can be decomposed into distinct chunks. The computation associated with the data chunks will only be efficient if the data chunks can be operated upon relatively independently.

Viewing the problem decomposition in terms of two distinct dimensions is somewhat artificial. A task decomposition implies a data decomposition and vice versa; hence, the two decompositions are really different facets of the same fundamental decomposition. We divide them into separate dimensions, however, because a problem decomposition usually proceeds most naturally by emphasizing one dimension of the decomposition over the other. By making them distinct, we make this design emphasis explicit and easier for the designer to understand.

3.1.3 Background for Examples

In this section, we give background information on some of the examples that are used in several patterns. It can be skipped for the time being and revisited later when reading a pattern that refers to one of the examples.

Medical imaging. PET (Positron Emission Tomography) scans provide an important diagnostic tool by allowing physicians to observe how a radioactive substance propagates through a patient's body. Unfortunately, the images formed from the distribution of emitted radiation are of low resolution, due in part to the scattering of the radiation as it passes through the body. It is also difficult to reason from the absolute radiation intensities, because different pathways through the body attenuate the radiation differently.

To solve this problem, models of how radiation propagates through the body are used to correct the images. A common approach is to build a Monte Carlo model, as described by Ljungberg and King [LK98]. Randomly selected points within the body are assumed to emit radiation (usually a gamma ray), and the trajectory of each ray is followed. As a particle (ray) passes through the body, it is attenuated by the different organs it traverses, continuing until the particle leaves the body and hits a camera model, thereby defining a full trajectory. To create a statistically significant simulation, thousands, if not millions, of trajectories are followed.

This problem can be parallelized in two ways. Because each trajectory is independent, it is possible to parallelize the application by associating each trajectory with a task. This approach is discussed in the Examples section of the *Task Decomposition* pattern. Another approach would be to partition the body into sections and assign different sections to different processing elements. This approach is discussed in the Examples section of the *Data Decomposition* pattern.

Linear algebra. Linear algebra is an important tool in applied mathematics: It provides the machinery required to analyze solutions of large systems of linear equations. The classic linear algebra problem asks, for matrix A and vector b, what values for x will solve the equation

$$A \cdot x = b \tag{3.1}$$

The matrix A in Eq. 3.1 takes on a central role in linear algebra. Many problems are expressed in terms of transformations of this matrix. These transformations are applied by means of a matrix multiplication

$$C = T \cdot A \tag{3.2}$$

If T, A, and C are square matrices of order N, matrix multiplication is defined such that each element of the resulting matrix C is

$$C_{i,j} = \sum_{k=0}^{N-1} T_{i,k} \cdot A_{k,j} \tag{3.3}$$

where the subscripts denote particular elements of the matrices. In other words, the element of the product matrix C in row i and column j is the dot product of the i-th row of T and the j-th column of A. Hence, computing each of the N^2 elements of C requires N multiplications and $N - 1$ additions, making the overall complexity of matrix multiplication $O(N^3)$.

There are many ways to parallelize a matrix multiplication operation. It can be parallelized using either a task-based decomposition (as discussed in the Examples section of the *Task Decomposition* pattern) or a data-based decomposition (as discussed in the Examples section of the *Data Decomposition* pattern).

Molecular dynamics. Molecular dynamics is used to simulate the motions of a large molecular system. For example, molecular dynamics simulations show how a

large protein moves around and how differently shaped drugs might interact with the protein. Not surprisingly, molecular dynamics is extremely important in the pharmaceutical industry. It is also a useful test problem for computer scientists working on parallel computing: It is straightforward to understand, relevant to science at large, and difficult to parallelize effectively. As a result, it has been the subject of much research [Mat94, PH95, Pli95].

The basic idea is to treat a molecule as a large collection of balls connected by springs. The balls represent the atoms in the molecule, while the springs represent the chemical bonds between the atoms. The molecular dynamics simulation itself is an explicit time-stepping process. At each time step, the force on each atom is computed and then standard classical mechanics techniques are used to compute how the force moves the atoms. This process is carried out repeatedly to step through time and compute a trajectory for the molecular system.

The forces due to the chemical bonds (the "springs") are relatively simple to compute. These correspond to the vibrations and rotations of the chemical bonds themselves. These are short-range forces that can be computed with knowledge of the handful of atoms that share chemical bonds. The major difficulty arises because the atoms have partial electrical charges. Hence, while atoms only interact with a small neighborhood of atoms through their chemical bonds, the electrical charges cause every atom to apply a force on every other atom.

This is the famous N-body problem. On the order of N^2 terms must be computed to find these nonbonded forces. Because N is large (tens or hundreds of thousands) and the number of time steps in a simulation is huge (tens of thousands), the time required to compute these nonbonded forces dominates the computation. Several ways have been proposed to reduce the effort required to solve the N-body problem. We are only going to discuss the simplest one: the *cutoff method.*

The idea is simple. Even though each atom exerts a force on every other atom, this force decreases with the square of the distance between the atoms. Hence, it should be possible to pick a distance beyond which the force contribution is so small that it can be ignored. By ignoring the atoms that exceed this cutoff, the problem is reduced to one that scales as $O(N \times n)$, where n is the number of atoms within the cutoff volume, usually hundreds. The computation is still huge, and it dominates the overall runtime for the simulation, but at least the problem is tractable.

There are a host of details, but the basic simulation can be summarized as in Fig. 3.2.

The primary data structures hold the atomic positions (`atoms`), the velocities of each atom (`velocity`), the forces exerted on each atom (`forces`), and lists of atoms within the cutoff distance of each atoms (`neighbors`). The program itself is a time-stepping loop, in which each iteration computes the short-range force terms, updates the neighbor lists, and then finds the nonbonded forces. After the force on each atom has been computed, a simple ordinary differential equation is solved to update the positions and velocities. Physical properties based on atomic motions are then updated, and we go to the next time step.

There are many ways to parallelize the molecular dynamics problem. We consider the most common approach, starting with the task decomposition (discussed


```
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
   vibrational_forces (N, atoms, forces)
   rotational_forces (N, atoms, forces)
   neighbor_list (N, atoms, neighbors)
   non_bonded_forces (N, atoms, neighbors, forces)
   update_atom_positions_and_velocities(
                  N, atoms, velocities, forces)
   physical_properties ( ... Lots of stuff ... )
end loop
```

Figure 3.2: Pseudocode for the molecular dynamics example

in the *Task Decomposition* pattern) and following with the associated data decomposition (discussed in the *Data Decomposition* pattern). This example shows how the two decompositions fit together to guide the design of the parallel algorithm.

3.2 THE *TASK DECOMPOSITION* PATTERN

Problem

How can a problem be decomposed into tasks that can execute concurrently?

Context

Every parallel algorithm design starts from the same point, namely a good understanding of the problem being solved. The programmer must understand which are the computationally intensive parts of the problem, the key data structures, and how the data is used as the problem's solution unfolds.

The next step is to define the tasks that make up the problem and the data decomposition implied by the tasks. Fundamentally, every parallel algorithm involves a collection of tasks that can execute concurrently. The challenge is to find these tasks and craft an algorithm that lets them run concurrently.

In some cases, the problem will naturally break down into a collection of independent (or nearly independent) tasks, and it is easiest to start with a *task-based decomposition.* In other cases, the tasks are difficult to isolate and the decomposition of the data (as discussed in the *Data Decomposition* pattern) is a better starting point. It is not always clear which approach is best, and often the algorithm designer needs to consider both.

Regardless of whether the starting point is a task-based or a data-based decomposition, however, a parallel algorithm ultimately needs tasks that will execute concurrently, so these tasks must be identified.

Forces

The main forces influencing the design at this point are flexibility, efficiency, and simplicity.

- **Flexibility.** Flexibility in the design will allow it to be adapted to different implementation requirements. For example, it is usually not a good idea to narrow the options to a single computer system or style of programming at this stage of the design.
- **Efficiency.** A parallel program is only useful if it scales efficiently with the size of the parallel computer (in terms of reduced runtime and/or memory utilization). For a task decomposition, this means we need enough tasks to keep all the PEs busy, with enough work per task to compensate for overhead incurred to manage dependencies. However, the drive for efficiency can lead to complex decompositions that lack flexibility.
- **Simplicity.** The task decomposition needs to be complex enough to get the job done, but simple enough to let the program be debugged and maintained with reasonable effort.

Solution

The key to an effective task decomposition is to ensure that the tasks are sufficiently independent so that managing dependencies takes only a small fraction of the program's overall execution time. It is also important to ensure that the execution of the tasks can be evenly distributed among the ensemble of PEs (the load-balancing problem).

In an ideal world, the compiler would find the tasks for the programmer. Unfortunately, this almost never happens. Instead, it must usually be done by hand based on knowledge of the problem and the code required to solve it. In some cases, it might be necessary to completely recast the problem into a form that exposes relatively independent tasks.

In a task-based decomposition, we look at the problem as a collection of distinct tasks, paying particular attention to

- The actions that are carried out to solve the problem. (Are there enough of them to keep the processing elements on the target machines busy?)
- Whether these actions are distinct and relatively independent.

As a first pass, we try to identify as many tasks as possible; it is much easier to start with too many tasks and merge them later on than to start with too few tasks and later try to split them.

Tasks can be found in many different places.

- In some cases, each task corresponds to a distinct call to a function. Defining a task for each function call leads to what is sometimes called a functional decomposition.

- Another place to find tasks is in distinct iterations of the loops within an algorithm. If the iterations are independent and there are enough of them, then it might work well to base a task decomposition on mapping each iteration onto a task. This style of task-based decomposition leads to what are sometimes called loop-splitting algorithms.
- Tasks also play a key role in data-driven decompositions. In this case, a large data structure is decomposed and multiple units of execution concurrently update different chunks of the data structure. In this case, the tasks are those updates on individual chunks.

Also keep in mind the forces given in the Forces section:

- **Flexibility.** The design needs to be flexible in the number of tasks generated. Usually this is done by parameterizing the number and size of tasks on some appropriate dimension. This will let the design be adapted to a wide range of parallel computers with different numbers of processors.
- **Efficiency.** There are two major efficiency issues to consider in the task decomposition. First, each task must include enough work to compensate for the overhead incurred by creating the tasks and managing their dependencies. Second, the number of tasks should be large enough so that all the units of execution are busy with useful work throughout the computation.
- **Simplicity.** Tasks should be defined in a way that makes debugging and maintenance simple. When possible, tasks should be defined so they reuse code from existing sequential programs that solve related problems.

After the tasks have been identified, the next step is to look at the data decomposition implied by the tasks. The *Data Decomposition* pattern may help with this analysis.

Examples

Medical imaging. Consider the medical imaging problem described in Sec. 3.1.3. In this application, a point inside a model of the body is selected randomly, a radioactive decay is allowed to occur at this point, and the trajectory of the emitted particle is followed. To create a statistically significant simulation, thousands, if not millions, of trajectories are followed.

It is natural to associate a task with each trajectory. These tasks are particularly simple to manage concurrently because they are completely independent. Furthermore, there are large numbers of trajectories, so there will be many tasks, making this decomposition suitable for a large range of computer systems, from a shared-memory system with a small number of processing elements to a large cluster with hundreds of processing elements.

With the basic tasks defined, we now consider the corresponding data decomposition—that is, we define the data associated with each task. Each task

needs to hold the information defining the trajectory. But that is not all: The tasks need access to the model of the body as well. Although it might not be apparent from our description of the problem, the body model can be extremely large. Because it is a read-only model, this is no problem if there is an effective shared-memory system; each task can read data as needed. If the target platform is based on a distributed-memory architecture, however, the body model will need to be replicated on each PE. This can be very time-consuming and can waste a great deal of memory. For systems with small memories per PE and/or with slow networks between PEs, a decomposition of the problem based on the body model might be more effective.

This is a common situation in parallel programming: Many problems can be decomposed primarily in terms of data or primarily in terms of tasks. If a task-based decomposition avoids the need to break up and distribute complex data structures, it will be a much simpler program to write and debug. On the other hand, if memory and/or network bandwidth is a limiting factor, a decomposition that focuses on the data might be more effective. It is not so much a matter of one approach being "better" than another as a matter of balancing the needs of the machine with the needs of the programmer. We discuss this in more detail in the *Data Decomposition* pattern.

Matrix multiplication. Consider the multiplication of two matrices ($C = A \cdot B$), as described in Sec. 3.1.3. We can produce a task-based decomposition of this problem by considering the calculation of each element of the product matrix as a separate task. Each task needs access to one row of A and one column of B. This decomposition has the advantage that all the tasks are independent, and because all the data that is shared among tasks (A and B) is read-only, it will be straightforward to implement in a shared-memory environment.

The performance of this algorithm, however, would be poor. Consider the case where the three matrices are square and of order N. For each element of C, N elements from A and N elements from B would be required, resulting in $2N$ memory references for N multiply/add operations. Memory access time is slow compared to floating-point arithmetic, so the bandwidth of the memory subsystem would limit the performance.

A better approach would be to design an algorithm that maximizes reuse of data loaded into a processor's caches. We can arrive at this algorithm in two different ways. First, we could group together the elementwise tasks we defined earlier so the tasks that use similar elements of the A and B matrices run on the same UE (see the *Group Tasks* pattern). Alternatively, we could start with the data decomposition and design the algorithm from the beginning around the way the matrices fit into the caches. We discuss this example further in the Examples section of the *Data Decomposition* pattern.

Molecular dynamics. Consider the molecular dynamics problem described in Sec. 3.1.3. Pseudocode for this example is shown again in Fig. 3.3.

Before performing the task decomposition, we need to better understand some details of the problem. First, the `neighbor_list()` computation is time-consuming.


```
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
    vibrational_forces (N, atoms, forces)
    rotational_forces (N, atoms, forces)
    neighbor_list (N, atoms, neighbors)
    non_bonded_forces (N, atoms, neighbors, forces)
    update_atom_positions_and_velocities(
                    N, atoms, velocities, forces)
    physical_properties ( ... Lots of stuff ... )
end loop
```

Figure 3.3: Pseudocode for the molecular dynamics example

The gist of the computation is a loop over each atom, inside of which every other atom is checked to determine whether it falls within the indicated cutoff volume. Fortunately, the time steps are very small, and the atoms don't move very much in any given time step. Hence, this time-consuming computation is only carried out every 10 to 100 steps.

Second, the `physical_properties()` function computes energies, correlation coefficients, and a host of interesting physical properties. These computations, however, are simple and do not significantly affect the program's overall runtime, so we will ignore them in this discussion.

Because the bulk of the computation time will be in `non_bonded_forces()`, we must pick a problem decomposition that makes that computation run efficiently in parallel. The problem is made easier by the fact that each of the functions inside the time loop has a similar structure: In the sequential version, each function includes a loop over atoms to compute contributions to the force vector. Thus, a natural task definition is the update required by each atom, which corresponds to a loop iteration in the sequential version. After performing the task decomposition, therefore, we obtain the following tasks.

- Tasks that find the vibrational forces on an atom
- Tasks that find the rotational forces on an atom
- Tasks that find the nonbonded forces on an atom
- Tasks that update the position and velocity of an atom
- A task to update the neighbor list for all the atoms (which we will leave sequential)

With our collection of tasks in hand, we can consider the accompanying data decomposition. The key data structures are the neighbor list, the atomic

coordinates, the atomic velocities, and the force vector. Every iteration that updates the force vector needs the coordinates of a neighborhood of atoms. The computation of nonbonded forces, however, potentially needs the coordinates of all the atoms, because the molecule being simulated might fold back on itself in unpredictable ways. We will use this information to carry out the data decomposition (in the *Data Decomposition* pattern) and the data-sharing analysis (in the *Data Sharing* pattern).

Known uses. Task-based decompositions are extremely common in parallel computing. For example, the distance geometry code DGEOM [Mat96] uses a task-based decomposition, as does the parallel WESDYN molecular dynamics program [MR95].

3.3 THE *DATA DECOMPOSITION* PATTERN

Problem

How can a problem's data be decomposed into units that can be operated on relatively independently?

Context

The parallel algorithm designer must have a detailed understanding of the problem being solved. In addition, the designer should identify the most computationally intensive parts of the problem, the key data structures required to solve the problem, and how data is used as the problem's solution unfolds.

After the basic problem is understood, the parallel algorithm designer should consider the tasks that make up the problem and the data decomposition implied by the tasks. Both the task and data decompositions need to be addressed to create a parallel algorithm. The question is not which decomposition to do. The question is which one to start with. A data-based decomposition is a good starting point if the following is true.

- The most computationally intensive part of the problem is organized around the manipulation of a large data structure.
- Similar operations are being applied to different parts of the data structure, in such a way that the different parts can be operated on relatively independently.

For example, many linear algebra problems update large matrices, applying a similar set of operations to each element of the matrix. In these cases, it is straightforward to drive the parallel algorithm design by looking at how the matrix can be broken up into blocks that are updated concurrently. The task definitions then follow from how the blocks are defined and mapped onto the processing elements of the parallel computer.

Forces

The main forces influencing the design at this point are flexibility, efficiency, and simplicity.

- **Flexibility.** Flexibility will allow the design to be adapted to different implementation requirements. For example, it is usually not a good idea to narrow the options to a single computer system or style of programming at this stage of the design.
- **Efficiency.** A parallel program is only useful if it scales efficiently with the size of the parallel computer (in terms of reduced runtime and/or memory utilization).
- **Simplicity.** The decomposition needs to be complex enough to get the job done, but simple enough to let the program be debugged and maintained with reasonable effort.

Solution

In shared-memory programming environments such as OpenMP, the data decomposition will frequently be implied by the task decomposition. In most cases, however, the decomposition will need to be done by hand, because the memory is physically distributed, because data dependencies are too complex without explicitly decomposing the data, or to achieve acceptable efficiency on a NUMA computer.

If a task-based decomposition has already been done, the data decomposition is driven by the needs of each task. If well-defined and distinct data can be associated with each task, the decomposition should be simple.

When starting with a data decomposition, however, we need to look not at the tasks, but at the central data structures defining the problem and consider whether they can they be broken down into chunks that can be operated on concurrently. A few common examples include the following.

- **Array-based computations.** Concurrency can be defined in terms of updates of different segments of the array. If the array is multidimensional, it can be decomposed in a variety of ways (rows, columns, or blocks of varying shapes).
- **Recursive data structures.** We can think of, for example, decomposing the parallel update of a large tree data structure by decomposing the data structure into subtrees that can be updated concurrently.

Regardless of the nature of the underlying data structure, if the data decomposition is the primary factor driving the solution to the problem, it serves as the organizing principle of the parallel algorithm.

When considering how to decompose the problem's data structures, keep in mind the competing forces.

- **Flexibility.** The size and number of data chunks should be flexible to support the widest range of parallel systems. One approach is to define chunks whose size and number are controlled by a small number of parameters. These parameters define *granularity knobs* that can be varied to modify the size of the data chunks to match the needs of the underlying hardware. (Note, however, that many designs are not infinitely adaptable with respect to granularity.)

  The easiest place to see the impact of granularity on the data decomposition is in the overhead required to manage dependencies between chunks. The time required to manage dependencies must be small compared to the overall runtime. In a good data decomposition, the dependencies scale at a lower dimension than the computational effort associated with each chunk. For example, in many finite difference programs, the cells at the boundaries between chunks, that is, the surfaces of the chunks, must be shared. The size of the set of dependent cells scales as the surface area, while the effort required in the computation scales as the volume of the chunk. This means that the computational effort can be scaled (based on the chunk's volume) to offset overheads associated with data dependencies (based on the surface area of the chunk).

- **Efficiency.** It is important that the data chunks be large enough that the amount of work to update the chunk offsets the overhead of managing dependencies. A more subtle issue to consider is how the chunks map onto UEs. An effective parallel algorithm must balance the load between UEs. If this isn't done well, some PEs might have a disproportionate amount of work, and the overall scalability will suffer. This may require clever ways to break up the problem. For example, if the problem clears the columns in a matrix from left to right, a column mapping of the matrix will cause problems as the UEs with the leftmost columns will finish their work before the others. A row-based block decomposition or even a block-cyclic decomposition (in which rows are assigned cyclically to PEs) would do a much better job of keeping all the processors fully occupied. These issues are discussed in more detail in the *Distributed Array* pattern.

- **Simplicity.** Overly complex data decompositions can be very difficult to debug. A data decomposition will usually require a mapping of a global index space onto a task-local index space. Making this mapping abstract allows it to be easily isolated and tested.

After the data has been decomposed, if it has not already been done, the next step is to look at the task decomposition implied by the tasks. The *Task Decomposition* pattern may help with this analysis.

Examples

Medical imaging. Consider the medical imaging problem described in Sec. 3.1.3. In this application, a point inside a model of the body is selected randomly, a radioactive decay is allowed to occur at this point, and the trajectory of the emitted particle is followed. To create a statistically significant simulation, thousands if not millions of trajectories are followed.

In a data-based decomposition of this problem, the body model is the large central data structure around which the computation can be organized. The model is broken into segments, and one or more segments are associated with each processing element. The body segments are only read, not written, during the trajectory computations, so there are no data dependencies created by the decomposition of the body model.

After the data has been decomposed, we need to look at the tasks associated with each data segment. In this case, each trajectory passing through the data segment defines a task. The trajectories are initiated and propagated within a segment. When a segment boundary is encountered, the trajectory must be passed between segments. It is this transfer that defines the dependencies between data chunks.

On the other hand, in a task-based approach to this problem (as discussed in the *Task Decomposition* pattern), the trajectories for each particle drive the algorithm design. Each PE potentially needs to access the full body model to service its set of trajectories. In a shared-memory environment, this is easy because the body model is a read-only data set. In a distributed-memory environment, however, this would require substantial startup overhead as the body model is broadcast across the system.

This is a common situation in parallel programming: Different points of view lead to different algorithms with potentially very different performance characteristics. The task-based algorithm is simple, but it only works if each processing element has access to a large memory and if the overhead incurred loading the data into memory is insignificant compared to the program's runtime. An algorithm driven by a data decomposition, on the other hand, makes efficient use of memory and (in distributed-memory environments) less use of network bandwidth, but it incurs more communication overhead during the concurrent part of computation and is significantly more complex. Choosing which is the appropriate approach can be difficult and is discussed further in the *Design Evaluation* pattern.

Matrix multiplication. Consider the standard multiplication of two matrices ($C = A \cdot B$), as described in Sec. 3.1.3. Several data-based decompositions are possible for this problem. A straightforward one would be to decompose the product matrix C into a set of row blocks (set of adjacent rows). From the definition of matrix multiplication, computing the elements of a row block of C requires the full A matrix, but only the corresponding row block of B. With such a data decomposition, the basic task in the algorithm becomes the computation of the elements in a row block of C.

An even more effective approach that does not require the replication of the full A matrix is to decompose all three matrices into submatrices or blocks. The basic task then becomes the update of a C block, with the A and B blocks being cycled among the tasks as the computation proceeds. This decomposition, however, is much more complex to program; communication and computation must

be carefully coordinated during the most time-critical portions of the problem. We discuss this example further in the *Geometric Decomposition* and *Distributed Array* patterns.

One of the features of the matrix multiplication problem is that the ratio of floating-point operations ($O(N^3)$) to memory references ($O(N^2)$) is small. This implies that it is especially important to take into account the memory access patterns to maximize reuse of data from the cache. The most effective approach is to use the block (submatrix) decomposition and adjust the size of the blocks so the problems fit into cache. We could arrive at the same algorithm by carefully grouping together the elementwise tasks that were identified in the Examples section of the *Task Decomposition* pattern, but starting with a data decomposition and assigning a task to update each submatrix seems easier to understand.

Molecular dynamics. Consider the molecular dynamics problem described in Sec. 3.1.3 and in the Examples section of the *Task Decomposition* pattern. This problem naturally breaks down into a task decomposition with a task being an iteration of the loop over atoms in each of the force computation routines.

Summarizing our problem and its task decomposition, we have the following:

- Tasks that find the vibrational forces on an atom
- Tasks that find the rotational forces on an atom
- Tasks that find the nonbonded forces on an atom
- Tasks that update the position and velocity of an atom
- A task to update the neighbor list for all the atoms (which we will leave sequential)

The key data structures are

- An array of atom coordinates, one element per atom
- An array of atom velocities, one element per atom
- An array of lists, one per atom, each defining the neighborhood of atoms within the cutoff distance of the atom
- An array of forces on atoms, one element per atom

An element of the velocity array is used only by the task owning the corresponding atom. This data does not need to be shared and can remain local to the task. Every task, however, needs access to the full array of coordinates. Thus, it will make sense to replicate this data in a distributed-memory environment or share it among UEs in a shared-memory environment.

More interesting is the array of forces. From Newton's third law, the force from atom i on atom j is the negative of the force from atom j on atom i. We can exploit this symmetry to cut the amount of computation in half as we accumulate the force terms. The values in the force array are not in the computation until the last steps in which the coordinates and velocities are updated. Therefore, the approach used is to initialize the entire force array on each PE and have the tasks accumulate partial sums of the force terms into this array. After all the partial force terms have completed, we sum all the PEs' arrays together to provide the final force array. We discuss this further in the *Data Sharing* pattern.

Known uses. Data decompositions are very common in parallel scientific computing. The parallel linear algebra library ScaLAPACK [Sca, BCC+97] uses block-based decompositions. The PLAPACK environment [vdG97] for dense linear algebra problems uses a slightly different approach to data decomposition. If, for example, an equation of the form $y = Ax$ appears, instead of first partitioning matrix A, the vectors y and x are partitioned in a natural way and then the induced partition on A is determined. The authors report better performance and easier implementation with this approach.

The data decomposition used in our molecular dynamics example is described by Mattson and Ravishanker [MR95]. More sophisticated data decompositions for this problem that scale better for large numbers of nodes are discussed by Plimpton and Hendrickson [PH95, Pli95].

3.4 THE *GROUP TASKS* PATTERN

Problem

How can the tasks that make up a problem be grouped to simplify the job of managing dependencies?

Context

This pattern can be applied after the corresponding task and data decompositions have been identified as discussed in the *Task Decomposition* and *Data Decomposition* patterns.

This pattern describes the first step in analyzing dependencies among the tasks within a problem's decomposition. In developing the problem's task decomposition, we thought in terms of tasks that can execute concurrently. While we did not emphasize it during the task decomposition, it is clear that these tasks do not constitute a flat set. For example, tasks derived from the same high-level operation in the algorithm are naturally grouped together. Other tasks may not be related in terms of the original problem but have similar constraints on their concurrent execution and can thus be grouped together.

In short, there is considerable structure to the set of tasks. These structures—these groupings of tasks—simplify a problem's dependency analysis. If a group shares a temporal constraint (for example, waiting on one group to finish filling a file before another group can begin reading it), we can satisfy that constraint once for the whole group. If a group of tasks must work together on a shared data structure, the required synchronization can be worked out once for the whole group.

If a set of tasks are independent, combining them into a single group and scheduling them for execution as a single large group can simplify the design and increase the available concurrency (thereby letting the solution scale to more PEs).

In each case, the idea is to define groups of tasks that share constraints and simplify the problem of managing constraints by dealing with groups rather than individual tasks.

Solution

Constraints among tasks fall into a few major categories.

- The easiest dependency to understand is a temporal dependency—that is, a constraint on the order in which a collection of tasks executes. If task A depends on the results of task B, for example, then task A must wait until task B completes before it can execute. We can usually think of this case in terms of data flow: Task A is blocked waiting for the data to be ready from task B; when B completes, the data flows into A. In some cases, A can begin computing as soon as data starts to flow from B (for example, pipeline algorithms as described in the *Pipeline* pattern).

- Another type of ordering constraint occurs when a collection of tasks must run at the same time. For example, in many data-parallel problems, the original problem domain is divided into multiple regions that can be updated in parallel. Typically, the update of any given region requires information about the boundaries of its neighboring regions. If all of the regions are not processed at the same time, the parallel program could stall or deadlock as some regions wait for data from inactive regions.

- In some cases, tasks in a group are truly independent of each other. These tasks do not have an ordering constraint among them. This is an important feature of a set of tasks because it means they can execute in any order, including concurrently, and it is important to clearly note when this holds.

The goal of this pattern is to group tasks based on these constraints, because of the following.

- By grouping tasks, we simplify the establishment of partial orders between tasks, since ordering constraints can be applied to groups rather than to individual tasks.

- Grouping tasks makes it easier to identify which tasks must execute concurrently.

For a given problem and decomposition, there may be many ways to group tasks. The goal is to pick a grouping of tasks that simplifies the dependency analysis. To clarify this point, think of the dependency analysis as finding and satisfying constraints on the concurrent execution of a program. When tasks share a set of constraints, it simplifies the dependency analysis to group them together.

There is no single way to find task groups. We suggest the following approach, keeping in mind that while one cannot think about task groups without considering the constraints themselves, at this point in the design, it is best to do so as abstractly as possible—identify the constraints and group tasks to help resolve them, but try not to get bogged down in the details.

- First, look at how the original problem was decomposed. In most cases, a high-level operation (for example, solving a matrix) or a large iterative program structure (for example, a loop) plays a key role in defining the decomposition. This is the first place to look for grouping tasks. The tasks that correspond to a high-level operation naturally group together.

  At this point, there may be many small groups of tasks. In the next step, we will look at the constraints shared between the tasks within a group. If the tasks share a constraint—usually in terms of the update of a shared data structure—keep them as a distinct group. The algorithm design will need to ensure that these tasks execute at the same time. For example, many problems involve the coordinated update of a shared data structure by a set of tasks. If these tasks do not run concurrently, the program could deadlock.

- Next, we ask if any other task groups share the same constraint. If so, merge the groups together. Large task groups provide additional concurrency to keep more PEs busy and also provide extra flexibility in scheduling the execution of the tasks, thereby making it easier to balance the load between PEs (that is, ensure that each of the PEs spends approximately the same amount of time working on the problem).

- The next step is to look at constraints between groups of tasks. This is easy when groups have a clear temporal ordering or when a distinct chain of data moves between groups. The more complex case, however, is when otherwise independent task groups share constraints between groups. In these cases, it can be useful to merge these into a larger group of independent tasks—once again because large task groups usually make for more scheduling flexibility and better scalability.

Examples

Molecular dynamics. This problem was described in Sec. 3.1.3, and we discussed its decomposition in the *Task Decomposition* and *Data Decomposition* patterns. We identified the following tasks:

- Tasks that find the vibrational forces on an atom
- Tasks that find the rotational forces on an atom
- Tasks that find the nonbonded forces on an atom
- Tasks that update the position and velocity of an atom

- A task to update the neighbor list for all the atoms (a single task because we have decided to leave this part of the computation sequential)

Consider how these can be grouped together. As a first pass, each item in the previous list corresponds to a high-level operation in the original problem and defines a task group. If we were to dig deeper into the problem, however, we would see that in each case the updates implied in the force functions are independent. The only dependency is the summation of the forces into a single force array.

We next want to see if we can merge any of these groups. Going down the list, the tasks in first two groups are independent but share the same constraints. In both cases, coordinates for a small neighborhood of atoms are read and local contributions are made to the force array, so we can merge these into a single group for bonded interactions. The other groups have distinct temporal or ordering constraints and therefore should not be merged.

Matrix multiplication. In the Examples section of the *Task Decomposition* pattern we discuss decomposing the matrix multiplication $C = A \cdot B$ into tasks, each corresponding to the update of one element in C. The memory organization of most modern computers, however, favors larger-grained tasks such as updating a block of C, as described in the Examples section of the *Data Decomposition* pattern. Mathematically, this is equivalent to grouping the elementwise update tasks into groups corresponding to blocks, and grouping the tasks this way is well suited to an optimum utilization of system memory.

3.5 THE *ORDER TASKS* PATTERN

Problem

Given a way of decomposing a problem into tasks and a way of collecting these tasks into logically related groups, how must these groups of tasks be ordered to satisfy constraints among tasks?

Context

This pattern constitutes the second step in analyzing dependencies among the tasks of a problem decomposition. The first step, addressed in the *Group Tasks* pattern, is to group tasks based on constraints among them. The next step, discussed here, is to find and correctly account for dependencies resulting from constraints on the order of execution of a collection of tasks. Constraints among tasks fall into a few major categories:

- Temporal dependencies, that is, constraints placed on the order in which a collection of tasks executes.
- Requirements that particular tasks must execute at the same time (for example, because each requires information that will be produced by the others).

- Lack of constraint, that is, total independence. Although this is not strictly speaking a constraint, it is an important feature of a set of tasks because it means they can execute in any order, including concurrently, and it is important to clearly note when this holds.

The purpose of this pattern is to help find and correctly account for dependencies resulting from constraints on the order of execution of a collection of tasks.

Solution

There are two goals to be met when identifying ordering constraints among tasks and defining a partial order among task groups.

- The ordering must be restrictive enough to satisfy all the constraints so that the resulting design is correct.
- The ordering should not be more restrictive than it needs to be. Overly constraining the solution limits design options and can impair program efficiency; the fewer the constraints, the more flexibility you have to shift tasks around to balance the computational load among PEs.

To identify ordering constraints, consider the following ways tasks can depend on each other.

- First look at the data required by a group of tasks before they can execute. After this data has been identified, find the task group that creates it and an ordering constraint will be apparent. For example, if one group of tasks (call it A) builds a complex data structure and another group (B) uses it, there is a sequential ordering constraint between these groups. When these two groups are combined in a program, they must execute in sequence, first A and then B.
- Also consider whether external services can impose ordering constraints. For example, if a program must write to a file in a certain order, then these file I/O operations likely impose an ordering constraint.
- Finally, it is equally important to note when an ordering constraint does not exist. If a number of task groups can execute independently, there is a much greater opportunity to exploit parallelism, so we need to note when tasks are independent as well as when they are dependent.

Regardless of the source of the constraint, we must define the constraints that restrict the order of execution and make sure they are handled correctly in the resulting algorithm. At the same time, it is important to note when ordering constraints are absent, since this will give valuable flexibility later in the design.

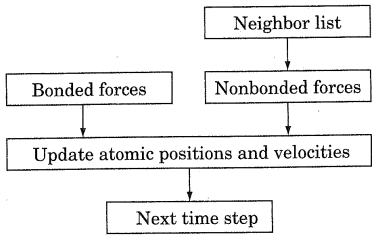

Figure 3.4: Ordering of tasks in molecular dynamics problem

Examples

Molecular dynamics. This problem was described in Sec. 3.1.3, and we discussed its decomposition in the *Task Decomposition* and *Data Decomposition* patterns. In the *Group Tasks* pattern, we described how to organize the tasks for this problem in the following groups:

- A group of tasks to find the "bonded forces" (vibrational forces and rotational forces) on each atom
- A group of tasks to find the nonbonded forces on each atom
- A group of tasks to update the position and velocity of each atom
- A task to update the neighbor list for all the atoms (which trivially constitutes a task group)

Now we are ready to consider ordering constraints between the groups. Clearly, the update of the atomic positions cannot occur until the force computation is complete. Also, the nonbonded forces cannot be computed until the neighbor list is updated. So in each time step, the groups must be ordered as shown in Fig. 3.4.

While it is too early in the design to consider in detail how these ordering constraints will be enforced, eventually we will need to provide some sort of synchronization to ensure that they are strictly followed.

3.6 THE *DATA SHARING* PATTERN

Problem

Given a data and task decomposition for a problem, how is data shared among the tasks?

Context

At a high level, every parallel algorithm consists of

- A collection of tasks that can execute concurrently (see the *Task Decomposition* pattern)

- A data decomposition corresponding to the collection of concurrent tasks (see the *Data Decomposition* pattern)
- Dependencies among the tasks that must be managed to permit safe concurrent execution

As addressed in the *Group Tasks* and *Order Tasks* patterns, the starting point in a dependency analysis is to group tasks based on constraints among them and then determine what ordering constraints apply to groups of tasks. The next step, discussed here, is to analyze how data is shared among groups of tasks, so that access to shared data can be managed correctly.

Although the analysis that led to the grouping of tasks and the ordering constraints among them focuses primarily on the task decomposition, at this stage of the dependency analysis, the focus shifts to the data decomposition, that is, the division of the problem's data into chunks that can be updated independently, each associated with one or more tasks that handle the update of that chunk. This chunk of data is sometimes called task-local data (or just local data), because it is tightly coupled to the task(s) responsible for its update. It is rare, however, that each task can operate using only its own local data; data may need to be shared among tasks in many ways. Two of the most common situations are the following.

- In addition to task-local data, the problem's data decomposition might define some data that must be shared among tasks; for example, the tasks might need to cooperatively update a large shared data structure. Such data cannot be identified with any given task; it is inherently global to the problem. This shared data is modified by multiple tasks and therefore serves as a source of dependencies among the tasks.
- Data dependencies can also occur when one task needs access to some portion of another task's local data. The classic example of this type of data dependency occurs in finite difference methods parallelized using a data decomposition, where each point in the problem space is updated using values from nearby points and therefore updates for one chunk of the decomposition require values from the boundaries of neighboring chunks.

This pattern discusses data sharing in parallel algorithms and how to deal with typical forms of shared data.

Forces

The goal of this pattern is to identify what data is shared among groups of tasks and determine how to manage access to shared data in a way that is both correct and efficient.

Data sharing can have major implications for both correctness and efficiency.

- If the sharing is done incorrectly, a task may get invalid data due to a race condition; this happens often in shared-address-space environments, where a task can read from a memory location before the write of the expected data has completed.

- Guaranteeing that shared data is ready for use can lead to excessive synchronization overhead. For example, an ordering constraint can be enforced by putting barrier operations[1] before reads of shared data. This can be unacceptably inefficient, however, especially in cases where only a small subset of the UEs are actually sharing the data. A much better strategy is to use a combination of copying into local data or restructuring tasks to minimize the number of times shared data must be read.

- Another source of data-sharing overhead is communication. In some parallel systems, any access to shared data implies the passing of a message between UEs. This problem can sometimes be mitigated by overlapping communication and computation, but this isn't always possible. Frequently, a better choice is to structure the algorithm and tasks so that the amount of shared data to communicate is minimized. Another approach is to give each UE its own copy of the shared data; this requires some care to be sure that the copies are kept consistent in value but can be more efficient.

  The goal, therefore, is to manage shared data enough to ensure correctness but not so much as to interfere with efficiency.

Solution

The first step is to identify data that is shared among tasks.

This is most obvious when the decomposition is predominantly a data-based decomposition. For example, in a finite difference problem, the basic data is decomposed into blocks. The nature of the decomposition dictates that the data at the edges of the blocks is shared between neighboring blocks. In essence, the data sharing was worked out when the basic decomposition was done.

In a decomposition that is predominantly task-based, the situation is more complex. At some point in the definition of tasks, it was determined how data is passed into or out of the task and whether any data is updated in the body of the task. These are the sources of potential data sharing.

After the shared data has been identified, it needs to be analyzed to see how it is used. Shared data falls into one of the following three categories.

- **Read-only.** The data is read but not written. Because it is not modified, access to these values does not need to be protected. On some distributed-memory systems, it is worthwhile to replicate the read-only data so each unit of execution has its own copy.

- **Effectively-local.** The data is partitioned into subsets, each of which is accessed (for read or write) by only one of the tasks. (An example of this would be an array shared among tasks in such a way that its elements are effectively partitioned into sets of task-local data.) This case provides some options for handling the dependencies. If the subsets can be accessed independently (as would normally be the case with, say, array elements, but not necessarily with list elements), then it is not necessary to worry about protecting access to this data. On distributed-memory systems, such data would usually be distributed among UEs, with each UE having only the data needed by its tasks. If necessary, the data can be recombined into a single data structure at the end of the computation.

- **Read-write.** The data is both read and written and is accessed by more than one task. This is the general case, and includes arbitrarily complicated situations in which data is read from and written to by any number of tasks. It is the most difficult to deal with, because any access to the data (read or write) must be protected with some type of exclusive-access mechanism (locks, semaphores, etc.), which can be very expensive.

Two special cases of read-write data are common enough to deserve special mention:

- **Accumulate.** The data is being used to accumulate a result (for example, when computing a reduction). For each location in the shared data, the values are updated by multiple tasks, with the update taking place through some sort of associative accumulation operation. The most common accumulation operations are sum, minimum, and maximum, but any associative operation on pairs of operands can be used. For such data, each task (or, usually, each UE) has a separate copy; the accumulations occur into these local copies, which are then accumulated into a single global copy as a final step at the end of the accumulation.

- **Multiple-read/single-write.** The data is read by multiple tasks (all of which need its initial value), but modified by only one task (which can read and write its value arbitrarily often). Such variables occur frequently in algorithms based on data decompositions. For data of this type, at least two copies are needed, one to preserve the initial value and one to be used by the modifying task; the copy containing the initial value can be discarded when no longer needed. On distributed-memory systems, typically a copy is created for each task needing access (read or write) to the data.

Examples

Molecular dynamics. This problem was described in Sec. 3.1.3, and we discussed its decomposition in the *Task Decomposition* and *Data Decomposition* patterns. We then identified the task groups (in the *Group Tasks* pattern) and considered temporal constraints among the task groups (in the *Order Tasks* pattern). We will ignore the temporal constraints for now and just focus on data sharing for the problem's final task groups:

- The group of tasks to find the "bonded forces" (vibrational forces and rotational forces) on each atom

[1] A barrier is a synchronization construct that defines a point in a program that a group of UEs must all reach before any of them are allowed to proceed.

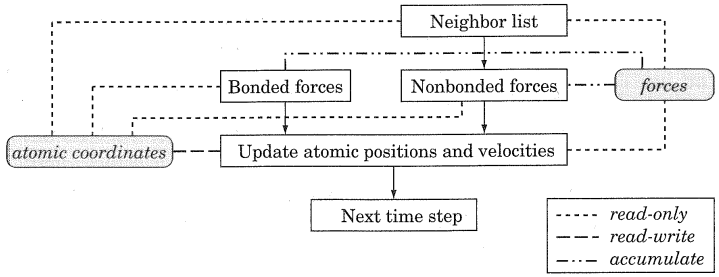

Figure 3.5: Data sharing in molecular dynamics. We distinguish between sharing for reads, read-writes, and accumulations.

- The group of tasks to find the nonbonded forces on each atom
- The group of tasks to update the position and velocity of each atom
- The task to update the neighbor list for all the atoms (which trivially constitutes a task group)

The data sharing in this problem can be complicated. We summarize the data shared between groups in Fig. 3.5. The major shared data items are the following.

- The atomic coordinates, used by each group.

  These coordinates are treated as read-only data by the bonded force group, the nonbonded force group, and the neighbor-list update group. This data is read-write for the position update group. Fortunately, the position update group executes alone after the other three groups are done (based on the ordering constraints developed using the *Order Tasks* pattern). Hence, in the first three groups, we can leave accesses to the position data unprotected or even replicate it. For the position update group, the position data belongs to the read-write category, and access to this data will need to be controlled carefully.

- The force array, used by each group except for the neighbor-list update.

  This array is used as read-only data by the position update group and as accumulate data for the bonded and nonbonded force groups. Because the position update group must follow the force computations (as determined using the *Order Tasks* pattern), we can put this array in the accumulate category for the force groups and in the read-only category for the position update group.

  The standard procedure for molecular dynamics simulations [MR95] begins by initializing the force array as a local array on each UE. Contributions to elements of the force array are then computed by each UE, with the precise terms computed being unpredictable because of the way the molecule folds in space. After all the forces have been computed, the local arrays are reduced into a single array, a copy of which is place on each UE (see the discussion of reduction in Sec. 6.4.2 for more information.)

- The neighbor list, shared between the nonbonded force group and the neighbor-list update group.

  The neighbor list is essentially local data for the neighbor-list update group and read-only data for the nonbonded force computation. The list can be managed in local storage on each UE.

3.7 THE *DESIGN EVALUATION* PATTERN

Problem

Is the decomposition and dependency analysis so far good enough to move on to the next design space, or should the design be revisited?

Context

At this point, the problem has been decomposed into tasks that can execute concurrently (using the *Task Decomposition* and *Data Decomposition* patterns) and the dependencies between them have been identified (using the *Group Tasks*, *Order Tasks*, and *Data Sharing* patterns). In particular, the original problem has been decomposed and analyzed to produce:

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

It is these four items that will guide the designer's work in the next design space (the *Algorithm Structure* patterns). Therefore, getting these items right and finding the best problem decomposition is important for producing a high-quality design.

In some cases, the concurrency is straightforward and there is clearly a single best way to decompose a problem. More often, however, multiple decompositions are possible. Hence, it is important before proceeding too far into the design process to evaluate the emerging design and make sure it meets the application's needs. Remember that algorithm design is an inherently iterative process, and designers should not expect to produce an optimum design on the first pass through the *Finding Concurrency* patterns.

Forces

The design needs to be evaluated from three perspectives.

- **Suitability for the target platform.** Issues such as number of processors and how data structures are shared will influence the efficiency of any design,

but the more the design depends on the target architecture, the less flexible it will be.

- **Design quality.** Simplicity, flexibility, and efficiency are all desirable—but possibly conflicting—attributes.

- **Preparation for the next phase of the design.** Are the tasks and dependencies regular or irregular (that is, are they similar in size, or do they vary)? Is the interaction between tasks synchronous or asynchronous (that is, do the interactions occur at regular intervals or highly variable or even random times)? Are the tasks aggregated in an effective way? Understanding these issues will help choose an appropriate solution from the patterns in the *Algorithm Structure* design space.

Solution

Before moving on to the next phase of the design process, it is helpful to evaluate the work so far from the three perspectives mentioned in the Forces section. The remainder of this pattern consists of questions and discussions to help with the evaluation.

Suitability for target platform. Although it is desirable to delay mapping a program onto a particular target platform as long as possible, the characteristics of the target platform do need to be considered at least minimally while evaluating a design. Following are some issues relevant to the choice of target platform or platforms.

How many PEs are available? With some exceptions, having many more tasks than PEs makes it easier to keep all the PEs busy. Obviously we can't make use of more PEs than we have tasks, but having only one or a few tasks per PE can lead to poor load balance. For example, consider the case of a Monte Carlo simulation in which a calculation is repeated over and over for different sets of randomly chosen data, such that the time taken for the calculation varies considerably depending on the data. A natural approach to developing a parallel algorithm would be to treat each calculation (for a separate set of data) as a task; these tasks are then completely independent and can be scheduled however we like. But because the time for each task can vary considerably, unless there are many more tasks than PEs, it will be difficult to achieve good load balance.

The exceptions to this rule are designs in which the number of tasks can be adjusted to fit the number of PEs in such a way that good load balance is maintained. An example of such a design is the block-based matrix multiplication algorithm described in the Examples section of the *Data Decomposition* pattern: Tasks correspond to blocks, and all the tasks involve roughly the same amount of computation, so adjusting the number of tasks to be equal to the number of PEs produces an algorithm with good load balance. (Note, however, that even in this case it might be advantageous to have more tasks than PEs. This might, for example, allow overlap of computation and communication.)

How are data structures shared among PEs? A design that involves large-scale or fine-grained data sharing among tasks will be easier to implement and more efficient if all tasks have access to the same memory. Ease of implementation depends on the programming environment; an environment based on a shared-memory model (all UEs share an address space) makes it easier to implement a design requiring extensive data sharing. Efficiency depends also on the target machine; a design involving extensive data-sharing is likely to be more efficient on a symmetric multiprocessor (where access time to memory is uniform across processors) than on a machine that layers a shared-memory environment over physically distributed memory. In contrast, if the plan is to use a message-passing environment running on a distributed-memory architecture, a design involving extensive data sharing is probably not a good choice.

For example, consider the task-based approach to the medical imaging problem described in the Examples section of the *Task Decomposition* pattern. This design requires that all tasks have read access to a potentially very large data structure (the body model). This presents no problems in a shared-memory environment; it is also no problem in a distributed-memory environment in which each PE has a large memory subsystem and there is plenty of network bandwidth to handle broadcasting the large data set. However, in a distributed-memory environment with limited memory or network bandwidth, the more memory-efficient algorithm that emphasizes the data decomposition would be required.

A design that requires fine-grained data-sharing (in which the same data structure is accessed repeatedly by many tasks, particularly when both reads and writes are involved) is also likely to be more efficient on a shared-memory machine, because the overhead required to protect each access is likely to be smaller than for a distributed-memory machine.

The exception to these principles would be a problem in which it is easy to group and schedule tasks in such a way that the only large-scale or fine-grained data sharing is among tasks assigned to the same unit of execution.

What does the target architecture imply about the number of UEs and how structures are shared among them? In essence, we revisit the preceding two questions, but in terms of UEs rather than PEs.

This can be an important distinction to make if the target system depends on multiple UEs per PE to hide latency. There are two factors to keep in mind when considering whether a design using more than one UE per PE makes sense.

The first factor is whether the target system provides efficient support for multiple UEs per PE. Some systems do provide such support, such as the Cray MTA machines and machines built with Intel processors that utilize hyperthreading. This architectural approach provides hardware support for extremely rapid context switching, making it practical to use in a far wider range of latency-hiding situations. Other systems do not provide good support for multiple UEs per PE.

For example, an MPP system with slow context switching and/or one processor per node might run much better when there is only one UE per PE.

The second factor is whether the design can make good use of multiple UEs per PE. For example, if the design involves communication operations with high latency, it might be possible to mask that latency by assigning multiple UEs to each PE so some UEs can make progress while others are waiting on a high-latency operation. If, however, the design involves communication operations that are tightly synchronized (for example, pairs of blocking send/receives) and relatively efficient, assigning multiple UEs to each PE is more likely to interfere with ease of implementation (by requiring extra effort to avoid deadlock) than to improve efficiency.

On the target platform, will the time spent doing useful work in a task be significantly greater than the time taken to deal with dependencies? A critical factor in determining whether a design is effective is the ratio of time spent doing computation to time spent in communication or synchronization: The higher the ratio, the more efficient the program. This ratio is affected not only by the number and type of coordination events required by the design, but also by the characteristics of the target platform. For example, a message-passing design that is acceptably efficient on an MPP with a fast interconnect network and relatively slow processors will likely be less efficient, perhaps unacceptably so, on an Ethernet-connected network of powerful workstations.

Note that this critical ratio is also affected by problem size relative to the number of available PEs, because for a fixed problem size, the time spent by each processor doing computation decreases with the number of processors, while the time spent by each processor doing coordination might stay the same or even increase as the number of processors increases.

Design quality. Keeping these characteristics of the target platform in mind, we can evaluate the design along the three dimensions of flexibility, efficiency, and simplicity.

Flexibility. It is desirable for the high-level design to be adaptable to a variety of different implementation requirements, and certainly all the important ones. The rest of this section provides a partial checklist of factors that affect flexibility.

- Is the decomposition flexible in the number of tasks generated? Such flexibility allows the design to be adapted to a wide range of parallel computers.
- Is the definition of tasks implied by the task decomposition independent of how they are scheduled for execution? Such independence makes the load balancing problem easier to solve.
- Can the size and number of chunks in the data decomposition be parameterized? Such parameterization makes a design easier to scale for varying numbers of PEs.

- Does the algorithm handle the problem's boundary cases? A good design will handle all relevant cases, even unusual ones. For example, a common operation is to transpose a matrix so that a distribution in terms of blocks of matrix *columns* becomes a distribution in terms of blocks of matrix *rows*. It is easy to write down the algorithm and code it for square matrices where the matrix order is evenly divided by the number of PEs. But what if the matrix is not square, or what if the number of rows is much greater than the number of columns and neither number is evenly divided by the number of PEs? This requires significant changes to the transpose algorithm. For a rectangular matrix, for example, the buffer that will hold the matrix block will need to be large enough to hold the larger of the two blocks. If either the row or column dimension of the matrix is not evenly divisible by the number of PEs, then the blocks will not be the same size on each PE. Can the algorithm deal with the uneven load that will result from having different block sizes on each PE?

Efficiency. The program should effectively utilize the available computing resources. The rest of this section gives a partial list of important factors to check. Note that typically it is not possible to simultaneously optimize all of these factors; design tradeoffs are inevitable.

- Can the computational load be evenly balanced among the PEs? This is easier if the tasks are independent, or if they are roughly the same size.
- Is the overhead minimized? Overhead can come from several sources, including creation and scheduling of the UEs, communication, and synchronization. Creation and scheduling of UEs involves overhead, so each UE needs to have enough work to do to justify this overhead. On the other hand, more UEs allow for better load balance.
- Communication can also be a source of significant overhead, particularly on distributed-memory platforms that depend on message passing. As we discussed in Sec. 2.6, the time to transfer a message has two components: latency cost arising from operating-system overhead and message start-up costs on the network, and a cost that scales with the length of the message. To minimize the latency costs, the number of messages to be sent should be kept to a minimum. In other words, a small number of large messages is better than a large number of small ones. The second term is related to the bandwidth of the network. These costs can sometimes be hidden by overlapping communication with computation.
- On shared-memory machines, synchronization is a major source of overhead. When data is shared between UEs, dependencies arise requiring one task to wait for another to avoid race conditions. The synchronization mechanisms used to control this waiting are expensive compared to many operations carried out by a UE. Furthermore, some synchronization constructs generate significant memory traffic as they flush caches, buffers, and other system resources to make sure UEs see a consistent view of memory. This extra memory

traffic can interfere with the explicit data movement within a computation. Synchronization overhead can be reduced by keeping data well-localized to a task, thereby minimizing the frequency of synchronization operations.

Simplicity. To paraphrase Einstein: Make it as simple as possible, but not simpler.

Keep in mind that practically all programs will eventually need to be debugged, maintained, and often enhanced and ported. A design—even a generally superior design—is not valuable if it is too hard to debug, maintain, and verify the correctness of the final program.

The medical imaging example initially described in Sec. 3.1.3 and then discussed further in the *Task Decomposition* and *Data Decomposition* patterns is an excellent case in point in support of the value of simplicity. In this problem, a large database could be decomposed, but this decomposition would force the parallel algorithm to include complex operations for passing trajectories between UEs and to distribute chunks of the database. This complexity makes the resulting program much more difficult to understand and greatly complicates debugging. The other approach, replicating the database, leads to a vastly simpler parallel program in which completely independent tasks can be passed out to multiple workers as they are read. All complex communication thus goes away, and the parallel part of the program is trivial to debug and reason about.

Preparation for next phase. The problem decomposition carried out with the *Finding Concurrency* patterns defines the key components that will guide the design in the *Algorithm Structure* design space:

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

Before moving on in the design, consider these components relative to the following questions.

How regular are the tasks and their data dependencies? Regular tasks are similar in size and effort. Irregular tasks would vary widely among themselves. If the tasks are irregular, the scheduling of the tasks and their sharing of data will be more complicated and will need to be emphasized in the design. In a regular decomposition, all the tasks are in some sense the same—roughly the same computation (on different sets of data), roughly the same dependencies on data shared with other tasks, etc. Examples include the various matrix multiplication algorithms described in the Examples sections of the *Task Decomposition, Data Decomposition,* and other patterns.

In an irregular decomposition, the work done by each task and/or the data dependencies vary among tasks. For example, consider a discrete-event simulation of a large system consisting of a number of distinct components. We might design a parallel algorithm for this simulation by defining a task for each component and having them interact based on the discrete events of the simulation. This would be a very irregular design in that there would be considerable variation among tasks with regard to work done and dependencies on other tasks.

Are interactions between tasks (or task groups) synchronous or asynchronous? In some designs, the interaction between tasks is also very regular with regard to time—that is, it is *synchronous.* For example, a typical approach to parallelizing a linear-algebra problem involving the update of a large matrix is to partition the matrix among tasks and have each task update its part of the matrix, using data from both its and other parts of the matrix. Assuming that all the data needed for the update is present at the start of the computation, these tasks will typically first exchange information and then compute independently. Another type of example is a *pipeline computation* (see the *Pipeline* pattern), in which we perform a multi-step operation on a sequence of sets of input data by setting up an assembly line of tasks (one for each step of the operation), with data flowing from one task to the next as each task accomplishes its work. This approach works best if all of the tasks stay more or less in step—that is, if their interaction is synchronous.

In other designs, the interaction between tasks is not so chronologically regular. An example is the discrete-event simulation described previously, in which the events that lead to interaction between tasks can be chronologically irregular.

Are the tasks grouped in the best way? The temporal relations are easy: Tasks that can run at the same time are naturally grouped together. But an effective design will also group tasks together based on their logical relationship in the overall problem.

As an example of grouping tasks, consider the molecular dynamics problem discussed in the Examples section of the *Group Tasks, Order Tasks,* and *Data Sharing* patterns. The grouping we eventually arrive at (in the *Group Tasks* pattern) is hierarchical: groups of related tasks based on the high-level operations of the problem, further grouped on the basis of which ones can execute concurrently. Such an approach makes it easier to reason about whether the design meets the necessary constraints (because the constraints can be stated in terms of the task groups defined by the high-level operations) while allowing for scheduling flexibility.

3.8 SUMMARY

Working through the patterns in the *Finding Concurrency* design space exposes the concurrency in your problem. The key elements following from that analysis are

- A task decomposition that identifies tasks that can execute concurrently
- A data decomposition that identifies data local to each task

- A way of grouping tasks and ordering the groups to satisfy temporal constraints
- An analysis of dependencies among tasks

A pattern language is traditionally described as a web of patterns with one pattern logically connected to the next. The output from the *Finding Concurrency* design space, however, does not fit into that picture. Rather, the goal of this design space is to help the designer create the design elements that together will lead into the rest of the pattern language.