Patterns for Finding Concurrency for Parallel Application Programs*

Berna L. Massingill† Timothy G. Mattson‡
Beverly A. Sanders§

Abstract

We are involved in an ongoing effort to develop a pattern language for parallel application programs. The pattern language consists of a set of patterns that guide the programmer through the entire process of developing a parallel program, including patterns that help £nd the concurrency in the problem, patterns that help £nd the appropriate algorithm structure to exploit the concurrency in parallel execution, and patterns describing lower-level implementation issues. The current version of the pattern language can be seen at `http://www.cise.ufl.edu/research/ParallelPatterns`.

In this contribution, we present patterns from the FindingConcurrency design space. These patterns form the starting point for novice parallel programmers and guide them through the process of identifying exploitable concurrency in a problem and designing a high-level algorithm to take advantage of this concurrency.

1 Introduction

1.1 Overview

Parallel hardware has been available for decades, and is becoming increasingly mainstream. Parallel software that fully exploits the hardware is much rarer, however, and mostly limited to the specialized area of supercomputing. We believe that part of the reason for this state of affairs is that most parallel programming environments, which focus on the implementation of concurrency rather than higher-level design issues, are simply too dif£cult for most programmers to risk using them.

†Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL; `blm@cise.ufl.edu` (current address: Department of Computer Science, Trinity University, San Antonio, TX; `bmassing@trinity.edu`).

‡Parallel Algorithms Laboratory, Intel Corporation; `timothy.g.mattson@intel.com`.

§Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL; `sanders@cise.ufl.edu`.

We are involved in an ongoing effort to design a pattern language for parallel application programs. The goal of the pattern language is to lower the barrier to parallel programming by guiding the programmer through the entire process of developing a parallel program. In our vision of parallel program development, the programmer brings into the process a good understanding of the actual problem to be solved, then works through the pattern language, eventually obtaining a detailed design or even working code. The pattern language is organized into four design spaces, which are visited in order.

- The FindingConcurrency design space includes high-level patterns that help £nd the concurrency in a problem and decompose it into a collection of tasks.
- The AlgorithmStructure design space contains patterns that help £nd an appropriate algorithm structure to exploit the concurrency that has been identi£ed.
- The SupportingStructures design space includes patterns that describe useful abstract data types and other supporting structures.
- The ImplementationMechanisms design space contains patterns that describe lower-level implementation issues.

The latter two design spaces (slightly stretching the typical notion of a pattern) might even include reusable code libraries or frameworks. We use a pattern format for all four levels so that we can address a variety of issues in a uni£ed way. The current, incomplete, version of the pattern language can be seen at `http://www.cise.ufl.edu/research/ParallelPatterns`. It consists of a collection of extensively hyperlinked documents, such that the designer can begin at the top level and work through the pattern language by following links.

In this paper, we present patterns from the FindingConcurrency design space. These patterns are used early in the design process, after the problem has been analyzed using standard software engineering techniques and the key data structures and computations are understood. They help programmers understand how to expose the exploitable concurrency in their problems. More speci£cally, these patterns help the programmer

- Identify the entities into which the problem will be decomposed.
- Determine how the entities depend on each other.
- Construct a coordination framework to manage the parallel execution of the entities.

These patterns collaborate closely with the AlgorithmStructure patterns, and one of their main functions is to help the programmer select an appropriate pattern in the AlgorithmStructure design space. Experienced designers might know how to do this immediately, in which case they could move directly to the patterns in the AlgorithmStructure design space.

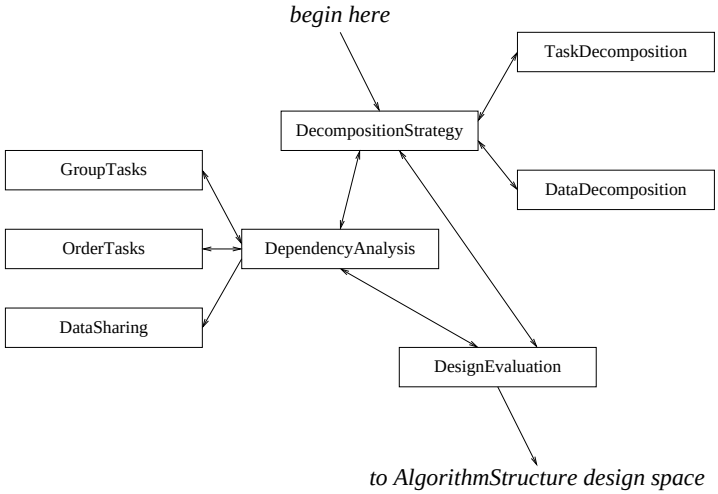

Figure 1: Organization of the FindingConcurrency design space.

1.2 Structure of the FindingConcurrency design space

The patterns in this design space are organized as illustrated in Figure 1. The main pathway through the patterns proceeds through three major patterns:

- **DecompositionStrategy:** This pattern helps the programmer decide whether the problem should be decomposed based on a data decomposition, a task decomposition, or a combination of the two.
- **DependencyAnalysis:** Once the entities into which the problem will be decomposed have been identi£ed, this pattern helps the programmer understand how they depend on each other.
- **DesignEvaluation:** This pattern is a consolidation pattern. It is used to evaluate the results of the other patterns in this design space and prepare the programmer for the next design space, the AlgorithmStructure design space.

Branching off from the DecompositionStrategy and DependencyAnalysis patterns are groups of patterns that help with problem decomposition and dependency analysis. We use double-headed arrows for most of the pathways in the £gure to indicate that one may need to move back and forth between the patterns repeatedly as the analysis proceeds. For example, in a dependency analysis, the programmer may group the tasks one way and then determine how this grouping affects the data that must be shared between the groups. This sharing may imply a different way to group the tasks, leading the programmer to revisit the tasks grouping. In general, one can expect working through these patterns to be an iterative process.

1.3 In this paper

The remainder of this paper consists of the complete text of most of the patterns of the FindingConcurrency design space. Each numbered major section represents one document in the collection of hyperlinked documents making up our pattern language; each document represents one pattern. To make the paper self-contained, we replace hyperlinks with text formatted like this and footnotes or citations.

2 The DecompositionStrategy Pattern

Intent:

This pattern addresses the question "How do you go about decomposing your problem into parts that can be run concurrently?"

Motivation:

Parallel programs let you solve bigger problems in less time. They do this by simultaneously solving different parts of the problem on different processors. This can only work if your problem contains exploitable concurrency, i.e., multiple activities or tasks that can take place at the same time.

Exposing concurrency requires decomposing the problem along two different dimensions:

- **Task decomposition.** Break the stream of instructions into multiple chunks called *tasks* that can execute simultaneously. To achieve reasonable runtime performance, tasks must execute with minimal need to interact; i.e., the overhead associated with managing dependencies must be small compared to the program's total execution time.

- **Data decomposition.** Determine how data interacts with the tasks. Some of the data will be modi£ed only within a task; i.e., it is local to each task. For such data, the algorithm designer must £gure out how to break up the data and properly associate it with the right tasks. Other data is modi£ed by multiple tasks; i.e., the data is global or shared between tasks. For shared data, the goal is to design the algorithm so that tasks don't get in each other's way as they work with the data.

Balancing the opposing forces of data and task decomposition occurs against the backdrop of two additional factors: ef£ciency and ¤exibility. The £nal program must effectively utilize the resources provided by the parallel computer. At the same time, parallel computers come in a variety of architectures, and you need enough ¤exibility to handle all the parallel computers you care about.

Note that in some cases the appropriate decomposition will be obvious; in others you will need to dig deeply into the problem to expose the concurrency. Sometimes you may even need to completely recast the problem and restructure how you think about its solution.

Applicability:

Use this pattern when

- You have determined that your problem is large and signi£cant enough that expending the effort to create a parallel program is worthwhile.
- You understand the key data structures of the problem and how they are used.
- You understand which parts of the problem are most compute-intensive. It is on these parts that you will focus your efforts.

Implementation:

The goal is to decompose your problem into relatively independent entities that can execute concurrently. As mentioned earlier, there are two dimensions to be considered:

- The **task-decomposition dimension** focuses on the operations that will take place within concurrently-executing entities. We refer to a set of operations that are logically grouped together as a *task*. For a task to be useful, 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. You need to decompose the problem's data into chunks that can be operated on relatively independently.

While the decomposition needs to address both the tasks and the data, the nature of the problem usually (but not always) suggests one decomposition or the other as the primary decomposition, and it is easiest to start with that one.

- A **data-based decomposition** is a good starting point if:
  - The most compute-intensive part of the problem manipulates 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 problems can be cast in terms of the multiplication of large matrices. Mathematically, each element of the product matrix is computed using the same set of operations. This suggests that an effective way to think about this problem is in terms of the decomposition of the matrices. We talk about this approach in more detail in the DataDecomposition pattern[1].

  Data-based decompositions tend to be more scalable (i.e., their performance scales with the number of processing elements), since memory is being decomposed.

[1] Section 4 of this paper.

- A **task-based decomposition** is a good starting point if:

  – It is natural to think about the problem in terms of a collection of independent (or nearly independent) tasks.

  For example, many problems can be considered in terms of a function that is evaluated repeatedly, with a slightly different set of conditions each time. We can associate each function evaluation with a task. If this is the case, you should start with a task-based decomposition, which is the subject of the TaskDecomposition pattern[2].

  If there are many nearly independent tasks, task-based decompositions tend to produce a design with a lot of ¤exibility, which is an advantage when later deciding how to allocate tasks to processing elements.

In some cases, you can view the problem in either way. For example, we earlier described a data decomposition of a matrix multiplication problem. You can also view this as a task-based decomposition — for example, by associating the update of each matrix column with a task. In cases where a clear decision cannot be made, the best approach is to try each decomposition and see which one is most effective at exposing lots of concurrency.

During the design process, you also need to keep in mind the following competing forces:

- **Flexibility.** Is your design abstract enough that you have suf£cient ¤exibility to adapt to different implementation requirements? For example, you don't want to narrow your options to a single computer system or style of programming at this stage of the design.

- **Ef£ciency.** A parallel program is only useful if it scales ef£ciently with the size of the parallel computer (in terms of reduced runtime and/or memory utilization). For the problem's decomposition, this means you need enough tasks to keep all the processing elements[3] (PEs) busy with enough work per task to compensate for overhead incurred to manage dependencies. The drive for ef£ciency can lead to complex decompositions that lack ¤exibility.

- **Simplicity.** Your decomposition needs to be complex enough to get the job done, but simple enough to let you debug and maintain your program in a reasonable amount of time.

Balancing these competing forces as you decompose your problem is dif£cult, and in all likelihood you will not get it right the £rst time.

Therefore, use an iterative decomposition strategy in which you decompose by the most obvious method (task or data) and then by the other method (data or task).

[2] Section 3 of this paper.

[3] Generic term used to reference a hardware element in a parallel computer that executes a stream of instructions.

Examples:

Medical imaging.

We will de£ne a single problem here, taken from the £eld of medical imaging, and then decompose it two different ways: in terms of tasks and in terms of data. The decompositions will be presented in the DataDecomposition and TaskDecomposition patterns; the discussion here will serve to de£ne the problem and then describe the way the two solutions interact.

An important diagnostic tool is to give a patient a radioactive substance and then watch how that substance propagates through the body by looking at the distribution of emitted radiation. Unfortunately, the images are of low resolution, due in part to the scattering of the radiation as it passes through the body. It is also dif£cult to reason from the absolute radiation intensities, since different pathways through the body attenuate the radiation differently.

To solve this problem, medical imaging specialists build models of how radiation propagates through the body and use these models to correct the images. A common approach is to build a Monte Carlo model. 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 de£ning a full trajectory. To create a statistically signi£cant simulation, thousands if not millions of trajectories are followed.

The problem can be parallelized in two ways. Since each trajectory is independent, it would be possible to parallelize the application by associating each trajectory with a task. This approach is discussed in the "Examples" section of the TaskDecomposition 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 DataDecomposition pattern.

As in many ray-tracing codes, there are no dependencies between trajectories, making the task-based decomposition the natural choice. By eliminating the need to manage dependencies, the task-based algorithm also gives the programmer plenty of ¤exibility later in the design process, when how to schedule the work on different processing elements becomes important.

The data decomposition, however, is much more effective at managing memory utilization. This is frequently the case with a data decomposition as compared to a task decomposition. Since memory is decomposed, data-decomposition algorithms also tend to be more scalable. These issues are important and point to the need to at least consider the types of platforms that will be supported by the £nal program. The need for portability drives one to make decisions about target platforms as late as possible. There are times, however, when delaying consideration of platform-dependent issues can lead one to choose a poor algorithm.

Parallel database.

As another example of a single problem that can be decomposed in multiple ways, consider a parallel database. One approach is to break up the database itself into mul-

tiple chunks. Multiple worker processes would handle the actual searching operations, each on the chunk of the database it "owns" and a single manager would receive search requests and forward each to the relevant worker to carry out the search.

A second approach for this parallel database problem would also use a manager and multiple workers but would keep the database intact in one logical location. The workers would be essentially identical and each would be able to work on any piece of the database.

Observe that the issues raised in this example are similar to those by the medical imaging example.

Iterative algorithms.

Many linear-algebra problems can be solved by repeatedly applying some operation to a large matrix or other array. Effective parallelizations of such algorithms are usually based on parallelizing each iteration (rather than, say, attempting to perform the iterations concurrently). For example, consider an algorithm that solves a system of linear equations $Ax = b$ (where A is a matrix and x and b are vectors) by calculating a sequence of approximations $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, and so forth, where for some function f, $x^{(k+1)} = f(x^{(k)})$. A typical parallelization would be structured as a sequential iteration (computing the $x^{(k)}$s in sequence), with each iteration (computing $x^{(k+1)} = f(x^{(k)})$ for some value of k) being computed in a way that exploits potential concurrency. For example, if each iteration requires a matrix multiplication, this operation can be parallelized using either a task-based decomposition (as discussed in the "Examples" section of the TaskDecomposition pattern) or a data-based decomposition (as discussed in the "Examples" section of the DataDecomposition pattern).

3 The TaskDecomposition Pattern

Intent:

This pattern addresses the question "How do you decompose a problem into tasks that can execute concurrently?"

Also Known As:

- Functional Decomposition.
- Task Parallelism.

Motivation:

This pattern addresses the issues raised during a primarily task-based decomposition. The key to an effective task decomposition is ensuring that the tasks are suf£ciently independent that maintaining 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 equally shared among the ensemble of processing elements (the so-called load-balancing problem).

Applicability:

Use this pattern when

- You have reviewed the DecompositionStrategy pattern[4] and decided to try a task-based decomposition of your problem.

Implementation:

It is very rare that a task-based decomposition can be carried out automatically. You have to do this by hand based on your knowledge of the problem and the code required to implement it.

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

- The actions that are carried out to solve your problem.
- Whether these actions are distinct and relatively independent.

As a £rst pass, 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. The key is that the individual tasks execute for the most part independently of each other.

You can £nd tasks in many different places:

- In some cases, each task corresponds to a distinct call to a function in your program. These function calls can be associated with the tasks, leading to what is sometimes called a “functional decomposition”.
- Another place to £nd tasks is in distinct iterations within an algorithm. If the iterations are independent and there are enough of them, then they map very well onto tasks in a task-based decomposition. 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.

This is only a partial list of where you can £nd tasks. Keep in mind the competing forces mentioned in the DecompositionStrategy pattern:

[4] Section 2 of this paper.

- **Flexibility.** Your design needs to be ¤exible in the number of tasks generated. This will let the design adapt to a wide range of parallel computers. Additional ¤exibility is gained if the de£nition of the tasks is independent of how they are scheduled for execution.

- **Ef£ciency.** There are two major ef£ciency issues to consider in your 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 must be large enough so that all the units of execution are busy with useful work throughout the computation.

- **Simplicity.** De£ne tasks to make debugging and maintenance simple. When possible, de£ne tasks so they reuse code from existing sequential programs that solve related problems.

Once you have your tasks, you need to look at the data decomposition implied by the tasks. The DataDecomposition pattern[5] may help you with this analysis.

Examples:

Medical imaging.

Consider the medical imaging problem described earlier (in the "Examples" section of the DecompositionStrategy pattern). 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 signi£cant simulation, thousands if not millions of trajectories are followed.

It is natural to associate each trajectory with a task. These tasks are particularly simple to manage concurrently since they are completely independent. Another important point is to make sure there are enough of them so we can support a range of computer systems, from those with a small number of processing elements to massively parallel computers.

With the basic tasks in hand, we can now consider the corresponding data decomposition and de£ne what portions of the problem space need to be associated with each task. In this case, each task needs to hold the information de£ning the trajectory at any point, but that is all. The more challenging issue is the model of the body. You can't deduce this from the description of the problem we've given, but it turns out that the body model can be huge. Since it is a read-only model, there is no problem if there is an effective shared-memory system, since each task can read data as needed. If the target system uses distributed memory, however, it may be necessary to replicate the body model on each processing element. This can be very time-consuming and can waste a great deal of memory. Thus, for such target systems a data-based decomposition, described in the "Examples" section of the DataDecomposition pattern, may work better.

[5] Section 4 of this paper.

Matrix multiplication.

Consider the standard multiplication of two matrices ($C = A \cdot B$). 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 likely be straightforward to implement in a shared-memory environment. In a distributed-memory environment, however, the requirement that each task have access to a row of A and a column of B may lead to excessive memory use and/or communication, so this decomposition may not be effective. See the "Examples" section of the DataDecomposition pattern for a discussion of other decompositions more suited to distributed-memory environments. (In fact, a data-based decomposition may be more effective for most current platforms — the above task-based decomposition works well only if access to memory elements is roughly uniform, which is not the case with cache-based computers.)

Known Uses:

Task-based decompositions are used in many applications. The distance geometry code (DGEOM) described in [Mattson96][2] uses a task based decomposition.

4 The DataDecomposition Pattern

Intent:

This pattern addresses the question "How do you decompose a problem's data into units that can be operated on relatively independently?"

Also Known As:

- Data Parallelism.

Motivation:

This pattern looks at the issues involved in decomposing data into units that can be updated concurrently.

For example, most 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 de£nitions then follow from how the blocks are de£ned and mapped onto the processing elements of the parallel computer.

Applicability:

Use this pattern when

- You have reviewed the DecompositionStrategy pattern[6] and decided to try a data-based decomposition of your problem.

Implementation:

Compilers are good at analyzing data dependencies and can in some cases automatically deduce a data decomposition. In most cases, however, you have to carry out the decomposition by hand.

If you have already carried out a task-based decomposition, the data decomposition is driven by the needs of each task. If well-de£ned and distinct data can be associated with each task, the decomposition should be simple.

If you are starting with a data decomposition, however, you need to look not at the tasks but at the central data structures de£ning your problem, considering whether they can they be broken down into chunks that can be operated on concurrently. A few common examples are

- Array-based computations: Concurrency can be de£ned in terms of updates of different segments of the array. If the array is multidimensional, observe that it can be decomposed in 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, the decomposition of the data serves as the organizing principle of your parallel algorithm.

As you consider how to decompose the problem's data structures, keep in mind the competing forces mentioned in the DecompositionStrategy pattern:

- **Flexibility.** The size and number of data chunks need to be ¤exible to support the widest range of parallel systems. Consider parameterizing the decomposition so it can be smoothly adjusted to £t the granularity of most parallel computers.

  If at all possible, you should de£ne chunks whose size and number are controlled by a small number of parameters. These parameters de£ne so-called "granularity knobs" that you can vary to modify the data decomposition to match the needs of the underlying hardware. (Note, however, that many designs are not in£nitely 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 £nite difference codes, the half-width of the £nite difference stencil de£nes a region of cells along the surface of each decomposed chunk. This surface region de£nes

[6] Section 2 of this paper.

the dependency between chunks. 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 you can scale the computational effort (based on the chunk's volume) to offset overheads associated with data dependencies (based on the surface area of the chunk).

- **Eff£ciency.** You need to make sure that the chunks are large enough so 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 units of execution[7]. An effective parallel algorithm must balance the load between units of execution. If this isn't done well, for example, several processing elements in the parallel computer may £nish their work before the others, 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 units of execution with the leftmost columns will £nish their work before the others. A row-based block decomposition or even a block-cyclic decomposition (in which rows are assigned cyclically to processing elements[8]) would do a much better job of keeping all the processors fully occupied.

- **Simplicity.** It may seem obvious, but many programmers have wasted countless hours trying to debug overly complex data decompositions. A data decomposition will usually require a mapping of a global index space onto a task-local index space. Make this mapping abstract so it can be easily isolated and tested.

Once you have your data decomposed, you need to look at the task decomposition implied by the tasks. The TaskDecomposition pattern may help you with this analysis.

Examples:

Medical imaging.

Consider the medical imaging problem described earlier (in the "Examples" section of the DecompositionStrategy pattern). 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 signi£cant simulation, thousands if not millions of trajectories are followed.

A task-based decomposition is a natural choice for this problem. Memory constraints, however, have motivated the development of data-based decompositions for this problem. When the memory of the underlying parallel hardware is distributed, it is advantageous to avoid replicating the huge body model on each processing element.

In a data-based decomposition, the body model is the large central data structure around which the computation can be organized. The model is broken into segments,

[7] Generic term for one of a collection of concurrently-executing entities, usually either processes or threads.

[8] Generic term used to reference a hardware element in a parallel computer that executes a stream of instructions.

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.

Once the data has been decomposed, you need to look at the tasks associated with each data segment. In this case, each trajectory passing through the data segment de£nes a task. The trajectories are initiated and propagated within a segment exactly as for the task-based approach. The difference occurs when a segment boundary is encountered. When this happens, the trajectory must be passed between segments. It is this transfer that de£nes the dependencies between data chunks.

Notice that this algorithm is more complex than one based on a task-based decomposition. Considerable effort can be required to implement the bookkeeping required to keep track of the set of trajectories as they move through the body model.

Matrix multiplication.

Consider the standard multiplication of two matrices ($C = A \cdot B$). In the "Examples" section of the TaskDecomposition pattern we discussed a task-based decomposition suitable for shared-memory environments but less so for distributed-memory environments. Several data-based decompositions are possible for this problem. A straightforward one would be to assign a row of the product matrix C to each processing element. From the de£nition of matrix multiplication, that means that each processing element would need the full A matrix, but only the corresponding row of B. With such a data decomposition, the basic task in our algorithm becomes the computation of a row of C. This still requires the replication of too much data (the full A matrix), however, so we might re£ne our algorithm so that we decompose all three matrices into blocks. The basic task then becomes the update of a C block, with the A and B blocks being cycled among the nodes as the computation proceeds. The result is the data-based decomposition discussed as an example of the GeometricDecomposition pattern[9]. Although such a block decomposition is more complex, it is nevertheless the approach used in practice since it is the most ef£cient.

Known Uses:

Data decompositions are very common in parallel scienti£c computing. The parallel linear algebra library ScaLAPACK is a good example.

5 The DependencyAnalysis Pattern

Intent:

This pattern addresses the question "After you have decomposed a problem into tasks, how do you analyze how they depend on each other?"

[9] A pattern in the AlgorithmStructure design space; see `http://www.cise.ufl.edu/research/ParallelPatterns`.

Motivation:

This pattern comes into play once you have decomposed a problem into tasks that can execute concurrently. In a few cases, not only can these tasks execute concurrently, but they are completely independent. For independent tasks, the programmer need only create the tasks and schedule them for ef£cient execution on the processing elements of the parallel computer.

More frequently, however, a problem's tasks are not independent; they in¤uence each other such that what happens in one task affects the execution of another task. We call these in¤uences between tasks *dependencies*. Dependencies take one of two basic forms:

- The most common form of dependency occurs when two or more tasks must share or exchange data as they execute. For example, in a data-based decomposition, the original problem domain is divided into multiple regions that can be updated in parallel. The update of any given region requires information from other regions (frequently, though not always, from the boundaries of its neighboring regions). This represents a *data-sharing dependency* between the regions.

- The second basic form of dependency is an *ordering constraint*. In other words, a collection of tasks may need to execute in a certain order. For example, a program may need to ensure that a complex data structure has been completely determined before a collection of tasks begins processing the data. Another form of ordering constraint occurs when a collection of tasks must run at the same time. For example, in the data decomposition example we discussed earlier, if all of the regions are not processed at the same time, the parallel program could stall (deadlock) as some regions wait for data from inactive regions.

Finding and managing dependencies is one of the most dif£cult jobs facing a parallel program designer.

Applicability:

Use this pattern when

- You have decomposed your problem into tasks that can execute concurrently, perhaps using the DecompositionStrategy pattern[10], and understand both the problem can be broken down into semi-independent tasks (its task decomposition) and how its data must be decomposed to support those tasks (its data decomposition).

Implementation:

The goal of a dependency analysis is to understand in detail how the tasks that make up your parallel program depend on each other. There are two kinds of dependencies:

[10] Section 2 of this paper.

- **Data-sharing dependencies**, which occur when two or more tasks must share or exchange data as they execute.
- **Ordering constraints**, which occur when tasks to execute in a certain order.

The dependencies between a problem's tasks have a major impact on both the ef£ciency and the complexity of the £nal program:

- The ef£ciency of a parallel program is proportional to the fraction of time spent making progress on a problem's computation. In this light, time spent communicating shared data or managing temporal constraints is wasted. Your dependencies must require little time to manage relative to the computation's time.
- Managing dependencies is a major source of complexity in a design and leads to many of the most serious errors in a parallel program. Probably the most dif£cult bugs to detect and £x are those that result from inconsistent sharing of data. Race conditions (situations in which program results depend on the relative order of task execution) frequently arise when the state of shared data is out of synch with task execution. For example, in an iterative algorithm, it is easy to commit synchronization errors that cause data from an earlier iteration to be incorrectly used in the present iteration. It is therefore important to manage dependencies so that these and other errors are easy to detect and £x.

Correctly analyzing dependencies among tasks is both dif£cult and crucial to the overall effectiveness of the design. There is no single way to accomplish this analysis, but we have found the following approach to be effective:

- First, identify how the tasks should be grouped together. Certain tasks may need to cooperatively update shared data structures, and therefore the algorithm design must ensure that they run together. On the other hand, a subset of tasks may be completely independent, but to help build a good scheduling algorithm to ef£ciently execute them, they should be grouped together. This and other issues pertaining to the grouping of tasks are discussed in the GroupTasks pattern[11].
- Next, identify any ordering constraints between groups of tasks. For example, if one group of tasks generates a matrix and another group uses that matrix, the second group must wait until the £rst group completes before it can execute. This process is discussed further in the OrderTasks pattern[12].
- Finally, analyze how tasks share data, both within and among groups. This process is discussed further in the DataSharing pattern[13].

It is not always possible to see whether decisions made at one step of this process will be effective until after you have worked through later steps. In fact, in many cases

[11] Section 6 of this paper.
[12] Section 7 of this paper.
[13] Section 8 of this paper.

you cannot fully understand whether a decomposition will be effective until after analyzing the resulting dependencies. Therefore, you should expect to iterate back and forth between the dependency and decomposition patterns, £rst carrying out a decomposition, analyzing its dependencies, and then reconsidering the decomposition. Even experienced parallel programmers may need to iterate though several cycles before getting it right.

Examples:

Molecular dynamics.

We will consider a single example as we look at each one of the dependency analysis design patterns. In the present pattern, we'll just introduce the problem and its decomposition. The dependency analysis itself will be described in the "Examples" sections of the individual dependency patterns.

Our example problem is to design a parallel molecular dynamics program. The theoretical background of molecular dynamics is interesting, but not really relevant to this discussion; we just need to understand the problem at its simplest level.

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 different-shaped drugs might interact with the protein. Not surprisingly, molecular dynamics is extremely important in the pharmaceutical industry. It turns out that molecular dynamics is important in computer science as well. It's a perfect test problem for computer scientists working on parallel computing: it's simple to understand, relevant to science at large, and very tough to effectively parallelize. Many papers have been written by computer scientists about parallel molecular dynamics algorithms (see the references in [Mattson94][1] for some of these papers).

So what is the basic molecular dynamics problem? The idea is to treat the 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, you compute the force on each atom, and then use standard classical mechanics 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 bounds themselves. These are short-range forces that can be computed with knowledge of the handful of atoms that share chemical bonds. What makes the molecular dynamics problem so dif£cult is the fact that the "balls" have partial electrical changes. Hence, while atoms interact with a small neighborhood of atoms through the chemical bonds, the electrical charge causes every atom to apply a force to every other atom.

This is the famous N-body problem. On the order of N^2 terms must be computed to get the long-range force. Since 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 long-range forces dominates the computation. Clever scientists have worked out elegant ways to reduce the effort required to solve the N-body problem.

We are only going to discuss the simplest of these tricks: the so-called cutoff method.

The idea is quite simple. Even though each atom exerts a force on every other atom, this force decreases as the distance between the atoms grows. Hence, it should be possible to pick a distance beyond which the force contribution can be ignored. That's the cutoff, and it reduces a problem that scales as $O(N^2)$ (where "$O(N)$" denotes "on the order of N") 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 with the following simpli£ed pseudo-code:

```
Int const N   // number of atoms

Array of real :: Atoms (3,N) //3D coordinates
Array of real :: Force (3,N) //force in each dimension
Array of lists :: 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)
    long_range_forces (N, Atoms, neighbors, Forces)
    update_atom_positions(N, Atoms, Forces)
    physical_properties ( ... Lots of stuff  ... )
end loop
```

There are a few details to mention, and then we can consider how this can be solved in parallel.

First, the `neighbor_list()` computation is time-consuming. The gist of the computation is a loop over each atom, inside which every other atom is checked to see if 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. For our discussion, we are not going to parallelize this computation.

Second, the `physical_properties()` function computes velocities, energies, correlation coef£cients, and a host of interesting physical properties. These computations, however, are not that involved and do not affect the overall parallel algorithm. Hence, we will ignore them in this discussion.

We are now ready to look at how to decompose this problem for execution on a parallel computer. In any parallel algorithm project, the £rst step is to decide where the most time-consuming computations are taking place. We have already discussed this and pointed out that the bulk of the time is in `long_range_forces()`. This means that whatever we do, we must pick a problem decomposition that makes that computation run ef£ciently in parallel.

While we won't discuss it in detail here (see [Mattson94][1] for details), each of the computations inside the time loop has a similar structure. Namely, they include a loop over atoms in which each loop iteration independently updates that atom's corre-

sponding array elements. So a natural task de£nition for each function is an iteration of this atom-loop: i.e., the update required by each atom.

Now let's look at the data decomposition. Each element of the array of atomic coordinates, `Atoms` (the array of atomic coordinates) is updated using its own data in all cases and coordinate data from a neighborhood of atoms in most cases. Elements of the `Forces` array are updated similarly (Newton's third law comes into play so when you compute the force of atom I on atom J you also get the negative of the force of atom J and atom I). Hence, these arrays need to be managed as shared data.

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

- Tasks that £nd the vibrational forces on an atom.
- Tasks that £nd the rotational forces on an atom.
- Tasks that £nd the long-range forces on an atom.
- Tasks that update the position of an atom.
- A task to update the neighbor list for all the atoms (which we will leave sequential).
- Shared arrays for the atomic coordinates and the forces.

Dependencies among these tasks and the associated data will be analyzed in the "Examples" sections of the dependency patterns (Group Tasks, OrderTasks, and DataSharing).

6 The GroupTasks Pattern

Intent:

This pattern addresses the question "How can you group the tasks that make up a problem decomposition in a way that simpli£es the job of managing dependencies?"

Motivation:

This pattern constitutes the £rst step in analyzing dependencies among the tasks of a problem decomposition. In developing the problem's task decomposition, we urged the designer to think of the problem 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 ¤at 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, you 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 the tasks in a group are independent in every way, it may simplify the design and increase the available concurrency (thereby letting you scale to more processing elements) to group them together.

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

Applicability:

Use this pattern when

- You have decomposed your problem into tasks that can execute concurrently (perhaps using the DecompositionStrategy pattern) and understand both the problem can be broken down into semi-independent tasks (its task decomposition) and how its data must be decomposed to support those tasks (its data decomposition).

Implementation:

Constraints among tasks fall into a few major categories, as follows.

- The easiest dependency to understand is a temporal dependency — i.e., a constraint placed 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 ¤ow: task *A* is blocked waiting for the data to be ready from task *B*; when *B* completes, the data ¤ows into *A*.

- Another form 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 (deadlock) as some regions wait for data from inactive regions.

- Finally, a more subtle ordering constraint occurs when tasks in a group are truly independent of each other. These tasks do not have an ordering constraint between 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 design pattern is to group tasks based on these constraints, because

- By grouping tasks, you simplify the establishment of partial orders between tasks since order constraints can be applied to the group rather than to individual tasks.

- Grouping tasks allows you to make clear 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 simpli£es the dependency analysis. To clarify this point, think of the dependency analysis as £nding and satisfying constraints on the concurrent execution of a program. When tasks share a set of constraints, it simpli£es the dependency analysis to group them together.

There is no single way to £nd task groups. We suggest the following approach, keeping in mind that while you 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 you decomposed the original problem. In most cases, a high-level operation (e.g., solving a matrix) or a large iterative program structure (e.g., a loop) plays a key role in de£ning the decomposition. This is the £rst place to look for grouping tasks. The tasks that correspond to a high-level operation naturally group together.

  At this point, you have many small groups of tasks. In the next step, you'll 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. Your algorithm design will need to ensure that these tasks execute at the same time. For example, many problems involve the concerted update of a shared data structure by a set of tasks. If these tasks do not run concurrently, the program could deadlock.

- Next, ask yourself if any other task groups share the same constraint. If so, merge the groups together. Large task groups give you ¤exibility in your design and make it easier for you to scale to large parallel systems. They also can simplify load balancing by giving you more ways to overlap concurrent execution of tasks within a group.

- 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 may be useful to merge these into a larger group of independent tasks — once again because large task groups usually make for more scheduling ¤exibility and better scalability.

Examples:

Molecular dynamics.

In the "Examples" section of the DependencyAnalysis pattern[14] we described the problem of designing a parallel molecular dynamics program. In that discussion, we de£ned the following tasks:

[14] Section 5 of this paper.

- Tasks that £nd the vibrational forces on an atom.
- Tasks that £nd the rotational forces on an atom.
- Tasks that £nd the long-range forces on an atom.
- Tasks that update the position 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).

Let's consider how these can be grouped together. As a £rst pass, each item in the above list corresponds to a high-level operation in the original problem and de£nes a task group. If you were to dig deeper into the involved functions, you'd see that in each case, the updates implied in the function are independent except for the need to manage the sum into the force array.

We next want to see if we can merge any of these groups. Going down the list, the tasks in £rst 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 can't be merged.

7 The OrderTasks Pattern

Intent:

This pattern addresses the question "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?"

Motivation:

This pattern constitutes the second step in analyzing dependencies among the tasks of a problem decomposition. The £rst step, addressed in the GroupTasks pattern[15], is to group tasks based on constraints among them. The next step, discussed here, is to help the designer £nd 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 (described in more detail in the GroupTasks pattern):

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

[15] Section 6 of this paper.

- Lack of constraint, i.e., total independence. While 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 design pattern is to help the designer £nd and correctly account for dependencies resulting from constraints on the order of execution of a collection of tasks.

Applicability:

Use this pattern when

- You have decomposed the problem into tasks and have decided where it makes sense to combine tasks into groups (as discussed in the GroupTasks pattern).

Implementation:

The goal of this pattern is to identify ordering constraints among groups of tasks and use them to de£ne a partial ordering among task groups. There are two goals to be met in de£ning this ordering:

- It must be restrictive enough to satisfy all the constraints, to be sure the resulting design is correct.
- It should not be more restrictive than it needs to be. Unneeded constraints can impair program ef£ciency; the fewer the constraints, the more ¤exibility you have to shift tasks around to balance the computational load between processing elements.

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. Once this data has been identi£ed, £nd the task group that created it and you will have an order constraint. For example, if one group of tasks builds a complex data structure and another group uses it, you need to specify a sequential order constraint between these groups. In other words, when you combine these two groups in a program, they need to follow a sequential ordering.
- Also consider whether external services can impose ordering constraints. For example, if a program must write to a £le in a certain order, then these £le I/O operations likely impose an ordering constraint.
- Finally, it is equally important to note when an order constraint does not exist. If a number of task groups can execute independently, you have a much greater opportunity to exploit parallelism, so be sure to note when tasks are independent, not just when they are dependent.

Regardless of the source of the constraint, your task as a designer is the same. You must de£ne the constraints that restrict the order of execution, and make sure they are handled correctly in the resulting algorithm. At the same time, you should note when ordering constraints are absent, since this will give you valuable ¤exibility later in your design.

Examples:

Molecular dynamics.

In the "Examples" section of the DependencyAnalysis pattern[16] we described the problem of designing a parallel molecular dynamics program. In the GroupTasks pattern, we further described how to organize the tasks in the following groups:

- A group of tasks to £nd the "bonded forces" (vibrational forces and rotational forces) on each atom.
- A group of tasks to £nd the long-range forces on each atom.
- A group of tasks to update the position 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 long-range forces cannot be computed until the neighbor list is updated. So in each time step, the groups must be ordered as shown in Figure 2.

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 primitive to ensure that they are strictly followed.

8 The DataSharing Pattern

Intent:

This pattern addresses the question "Given a way of decomposing a problem into tasks, how is data shared among the tasks?"

Motivation:

This pattern constitutes the third step in analyzing dependencies among the tasks of a problem decomposition. The £rst and second steps, addressed in the GroupTasks and OrderTasks patterns[17], are to group tasks based on constraints among them and then determine what ordering constraints apply to groups of tasks. The next step, discussed

[16] Section 5 of this paper.
[17] Sections 6 and 7 of this paper.

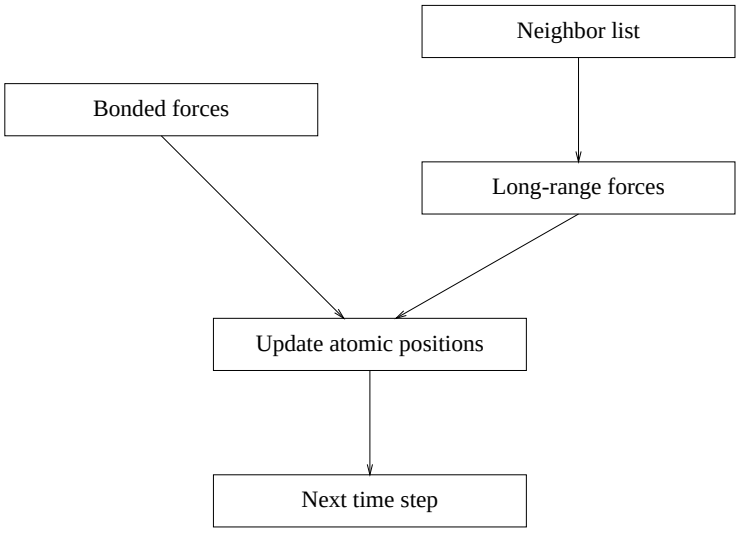

Figure 2: Ordering of tasks in molecular dynamics problem.

here, is to analyze how data is shared among groups of tasks, so that access to shared data can be managed correctly.

The £rst two steps of the dependency analysis have focused on how the original problem's computation was divided into tasks (the task decomposition). This step also takes into consideration the associated 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), since 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 may de£ne some data that must be shared among tasks; for example, the tasks may need to cooperatively update a large shared data structure. Such data cannot be identi£ed with any given task; it is inherently global to the problem. This shared data is modi£ed by multiple tasks and therefore serves as a source of dependencies between 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 £nite 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.

Applicability:

Use this pattern when

- You have decomposed the problem in terms of both tasks and data (task decomposition and data decomposition), have decided how to combine the tasks into groups (as discussed in the GroupTasks pattern), and have determined what ordering constraints apply among groups (as discussed in the OrderTasks pattern).

Implementation:

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

Data sharing can have major implications for both the correctness and the ef£ciency of your program:

- If the sharing is done incorrectly, a task may get invalid data; 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, you can in many cases force a desired order by putting barrier operations before reads of shared data, but this can be very ineff£cient if many units of execution[18] (UEs) wait at a barrier that is only needed to properly synchronize execution of a few UEs. 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 units of execution. You can sometimes avoid this problem by overlapping communication and computation, but this isn't always possible. Frequently, a better choice is to structure your algorithm and the way you de£ne tasks so that the amount of shared data to communicate is minimized. Another approach is to give each unit of execution 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 ef£cient.

The goal, therefore, is to manage shared data enough to ensure correctness but not so much as to interfere with ef£ciency. We suggest the following approach to determining what data is shared and how to manage it:

[18] Generic term for one of a collection of concurrently-executing entities, usually either processes or threads.

- The first step is to identify data that is shared between tasks. The data sharing implied by your algorithm is closely connected to the basic way you decomposed your problem.

  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, you worked out the data sharing 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, you needed to define how data passed into or out of the task and whether any data was updated in the body of the task. These are your sources of potential data sharing.

- Once you have identified any data that is shared, you need to understand how the data will be used. Shared data falls into one of the following three categories:

  - **Read-only.** The data is read but not written. Since 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 gives the programmer some options. If the subsets can accessed independently (as would normally be the case with, say, array elements, but not necessarily with list elements), then the programmer need not 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, since 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 (i.e., is being used to compute 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 cases

for the accumulation operations are sum, minimum, and maximum, but any associative pairwise operation will do. 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 £nal step at the end of the computation.

– **Multiple-read/single-write.** The data is read by multiple tasks (all of which need its initial value) but modi£ed by only 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 at the end of the computation. On distributed-memory systems, typically a copy is created for each task needing access (read or write) to the data.

Examples:

Molecular dynamics.

In the "Examples" section of the DependencyAnalysis pattern[19] we described the problem of designing a parallel molecular dynamics program. We then identi£ed the task groups (in the GroupTasks pattern) and considered temporal constraints between the task groups (in the OrderTasks pattern). We will ignore the temporal constraints for now and just focus on data sharing for the problem's £nal task groups:

- The group of tasks to £nd the "bonded forces" (vibrational forces and rotational forces) on each atom.
- The group of tasks to £nd the long-range forces on each atom.
- The group of tasks to update the position of each atom.
- The task to update the neighbor list for all the atoms (which trivially constitutes a task group).

When you analyze the computations taking place with each of these groups, you £nd the following shared data:

- The atomic coordinates, used by each group.

  These coordinates are treated as read-only data by the bonded force group, the long-range 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 OrderTasks pattern). Hence, in the £rst 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.

[19] Section 5 of this paper.

- 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 long-range force groups. Since the position update group must follow the force computations (as determined using the OrderTasks 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 neighbor list, shared between the long-range force group and the neighbor-list update group.

  This list is used by the long-range-force and neighbor-list update groups. It is essentially-local data for the neighbor-list update group and read-only data for the long-range force computation.

9 The DesignEvaluation Pattern

Intent:

In this pattern, we evaluate the design so far, and decide whether to revisit the design or move on to the next design space.

Motivation:

The patterns in the FindingConcurrency design space have helped the designer expose the concurrency in his or her problem. In particular, the original problem has been analyzed to produce:

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

We will use this information in the next design space — the AlgorithmStructure design space[20] — to construct an algorithm that can exploit this concurrency in a parallel program.

In some cases, the concurrency is straightforward and there is clearly a best way to decompose a problem to expose it. More commonly, however, there are many ways to decompose a problem into tasks. Choosing one may require tradeoffs between three criteria of a good design: simplicity, ¤exibility, and ef£ciency. Unfortunately, there is no foolproof way to be sure that you have de£ned the right set of tasks or even that the tasks have been correctly grouped. The design process is inherently iterative. This pattern will help you decide evaluate your design and decide whether to move on to the next design space or revisit the decomposition.

[20] The second major part of our pattern language, available as a collection of documents linked from `http://www.cise.ufl.edu/research/ParallelPatterns`.

Applicability:

Use this pattern when

- You have decomposed the problem into tasks that can execute concurrently (using the DecompositionStrategy pattern[21]) and determined their dependencies (using the DependencyAnalysis pattern[22]).

Implementation:

This pattern has two goals: to evaluate the design so far (with the possible result that the programmer decides to revisit and possible revise decisions made thus far) and to prepare for the next phase of the design process. We therefore describe how to evaluate the design from three perspectives: suitability for the target platform, design quality, and preparation for the next phase of the design.

Suitability for Target Platform.

While 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 your design. Below are some issues relevant to the choice of target platform or platforms.

How many processing elements are available?

With some exceptions, having many more tasks than processing elements[23] (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[24]. 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 since the time for each task can vary considerably, unless there are many more tasks than PEs it will be dif£cult to achieve good load balance.

The exceptions to this rule are designs in which the number of tasks can be adjusted to £t 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 DataDecomposition pattern[25]: Tasks correspond to

[21] Section 2 of this paper.

[22] Section 5 of this paper.

[23] Generic term used to reference a hardware element in a parallel computer that executes a stream of instructions.

[24] Load balance is the degree to which work is evenly distributed among available PEs. A parallel program executes most quickly when it is perfectly load balanced; that is, when work is divided among PEs such that all PEs complete their assigned tasks at the same time.

[25] Section 4 of this paper.

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, if for example that would allow overlap of computation and communication.)

HOW ARE DATA STRUCTURES SHARED AMONG PROCESSING ELEMENTS?

A design that involves large-scale or £ne-grained data sharing among tasks will be easier to implement and more ef£cient if all tasks have access to the same memory. Ease of implementation depends on programming environment; an environment based on shared-memory model (all units of execution[26] share an address space) makes it easier to implement a design requiring extensive data sharing. Ef£ciency depends also on the target machine; a design involving extensive data-sharing is likely to be more ef£cient 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 you plan to implement your design using a message-passing environment running on a distributed-memory architecture, a design involving extensive data sharing is likely not a good choice.

For example, consider the task-based approach to medical imaging problem described in the "Examples" section of the TaskDecomposition pattern. This design requires that all tasks have read access to a potentially very large data structure (the body model), which presents no problems in a shared-memory environment but in a distributed-memory environment can require prohibitive amounts of memory or communication.

A design that requires £ne-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 ef£cient 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 only exceptions to these principles would be problems in which it is easy to group and schedule tasks in such a way that the only large-scale or £ne-grained data sharing is among tasks assigned to the same unit of execution.

WHAT DOES THE TARGET ARCHITECTURE IMPLY ABOUT THE NUMBER OF UNITS OF EXECUTION AND HOW STRUCTURES ARE SHARED AMONG THEM?

In essence, this question asks you to revisit the preceding two questions, but in terms of units of execution (UEs) rather than processing elements (PEs). This can be an important distinction to make if the target system supports multiple UEs per PE, particularly if the target system emphasizes the use of multitasking to hide latency (an example of such a system is the Tera machine).

There are two factors to keep in mind when considering whether a design using more than one UE per PE makes sense.

[26] Generic term for one of a collection of concurrently-executing entities, usually either processes or threads.

The £rst factor is whether the target system provides ef£cient support for multiple UEs per PE. Some systems do provide such support (an example is the Tera machine, which was designed to provide ef£cient support for many more threads (UEs) than processors (PEs)). Other systems do not (an example is an MPP system with one processor per node and slow context-switching, where multiple processes (UEs) per processor (PE) are likely to be ineff£cient).

The second factor is whether the design can make good use of multiple UEs per PE. For example, if the design involves coordination operations with high latency, it might be possible to mask that latency by assigning multiple UEs to each PE. If however the design involves coordination operations that are tightly synchronized (e.g., pairs of blocking send/receives) and relatively ef£cient, 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 ef£ciency.

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 managing data dependencies ("coordination" — i.e., synchronization or communication among processing elements): The higher the ratio, the more ef£cient 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 ef£cient on an MPP with a fast interconnect network and relatively slow processors will likely be less ef£cient, perhaps unacceptably so, on an Ethernet-connected network of powerful workstations.

Note also that this critical ratio is also frequently affected by problem size relative to the number of available processing elements, since for a given 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 may 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 ¤exibility, ef£ciency, and simplicity.

FLEXIBILITY.

You would like your high level design to be adaptable to a variety of different implementation requirements, and certainly all the ones that you care about. The following is a partial checklist of factors that affect ¤exibility.

- Is your decomposition ¤exible in the number of tasks generated? Such ¤exibility allows your design to be adapted to a wide range of parallel computers.

- Is the de£nition of tasks implied by your 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 your data decomposition be parameterized? Such parameterization makes a design easier to scale for varying numbers of processing elements.

- Does your algorithm handle the problem's boundary cases? A good design will handle all relevant case, even unusual ones.

  For example, a common operation is to transpose a matrix such that a distribution in terms of blocks of matrix *columns* becomes a distribution in terms of blocks of matrix *rows*. It's easy to write down the algorithm and code it for square matrices where the matrix order is evenly divided by the number of processing elements. 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 nodes? This requires signi£cant 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 nodes, then the blocks will not be the same size on each node. Can your algorithm deal with the uneven load that will result from having different block sizes on each node?

EFFICIENCY.

You would like your program to effectively utilize the available computing resources. The following is 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 processing elements?

  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 thread creation and scheduling, communication, and synchronization.

  Thread creation and scheduling involve overhead, so you should make sure that each thread has enough work to do to justify this overhead. On the other hand, more threads allow for better load balance.

  Communication can also be a source of signi£cant overhead, particularly in platforms that use message-passing; message transfer typically involves both overhead due to kernel calls and latency due to the time it takes the message to travel over the network. While network latency can sometimes be hidden by overlapping it with computation, to minimize the overhead due to kernel calls, the number of messages to be sent should be minimized. Even on shared-memory multiprocessors, data should be localized as much as possible.

> Synchronization is required whenever a dependency requires one task to wait for another — either because the result is needed, or to avoid race conditions. Designs that minimize dependencies may reduce synchronization overhead.

SIMPLICITY.

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

Keep in mind that you will ultimately need to debug any program you write. A design — even a generally superior design — will not do you any good if you cannot debug, maintain, and verify the correctness of the £nal program.

The medical imaging example given in the DecompositionStrategy pattern 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 a simulation point from one database chunk to another. This complexity makes the resulting program much more dif£cult 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/or reason about.

Preparation for Next Phase.

The problem decomposition carried out with the FindingConcurrency patterns serves to prepare for the next design space. Many of the key issues that determine a high-quality problem decomposition cannot be described, let alone resolved, until you start working with the AlgorithmStructure patterns. However, here are some key issues to keep in mind as you enter the next design space.

HOW REGULAR ARE THE TASKS AND THEIR DATA DEPENDENCIES?

In other words, do they vary widely among themselves? If so, the scheduling of the tasks and their sharing of data may be an important issue. 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 TaskDecomposition and DataDecomposition 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 de£ning 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?

(Note this question is closely related to that of regular versus irregular.)

In some designs, the interaction between tasks is also very regular with regard to time — i.e., 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 £rst exchange information and then compute independently. Another type of example is a *pipeline computation*, 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 ¤owing 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 — i.e., 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 the discussion of regular versus irregular), in which the events that lead to interaction between tasks may 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 DependencyAnalysis pattern[27]. The grouping we eventually arrive at (in the GroupTasks pattern [28]) 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 (since the constraints can be stated in terms of the task groups de£ned by the high-level operations) while allowing for scheduling ¤exibility.

Acknowledgments

We thank Alan O'Callaghan for his helpful and insightful comments during the PLoP shepherding process and the members of our PLoP workshop group for their additional comments. We also thank Intel Corporation, the National Science Foundation (grant #9704697), and the Air Force Of£ce of Scienti£c Research (grant #4514209-12) for their £nancial support.

[27] Section 5 of this paper.
[28] Section 6 of this paper.

References

[1] T. G. Mattson. The ef£ciency of Linda for general purpose scienti£c programming. *Scienti£c Programming*, 3:61–71, 1994.

[2] T.G. Mattson. Scienti£c computation. In A. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw Hill, 1996.