Some Algorithm Structure and Support Patterns for Parallel Application Programs[*]

Berna L. Massingill[†] Timothy G. Mattson[‡]

Beverly A. Sanders[§]

Abstract

We are developing a pattern language to guide the programmer through the entire process of developing a parallel application program. The pattern language includes patterns that help find the concurrency in the problem, patterns that help find 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 paper, we outline the overall structure of the pattern language and present two groups of selected patterns, one group chosen from the subset of patterns that represent different strategies for exploiting concurrency once it has been identified and one group chosen from the subset of patterns that represent commonly-used computational and data structures.

1 Introduction

1.1 Overview

We are developing 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 three *design spaces*, each corresponding to one major phase in the design-and-development process:

[*]Copyright © 2002, Berna L. Massingill. Permission is granted to copy for the PLoP 2002 conference. All other rights reserved.

[†]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`.

- The *FindingConcurrency* design space includes high-level patterns that help find the concurrency in a problem and decompose it into a collection of tasks. These patterns, presented in [MMS00], help the designer determine (i) how to decompose the problem into tasks that can execute concurrently, (ii) which data is local to the tasks and which is shared among tasks, and (iii) what ordering and data-dependency constraints exist among tasks.

- The *AlgorithmStructure* design space contains patterns that help find an algorithm structure to exploit the concurrency that has been identified. These patterns, most of which are presented in [MMS99] and this paper, capture recurring solutions to the problem of turning problems into parallel algorithms; with one exception, each pattern represents a high-level strategy for exploiting available concurrency. (The exception is *ChooseStructure* (Section 3.1 of this paper), which helps the designer select an appropriate pattern from among the other patterns in this design space.) Examples of patterns in this design space are *EmbarrassinglyParallel* [MMS99] and *GeometricDecomposition* [MMS99].

- The *SupportingStructures* design space includes patterns that represent an intermediate stage between the problem-oriented patterns of the *AlgorithmStructure* design space and the APIs needed to implement them.

The current 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 outline the *FindingConcurrency*, *AlgorithmStructure*, and *SupportingStructures* design spaces and present the complete text of selected patterns from the latter two design spaces: *AlgorithmStructure* patterns *ChooseStructure*, *DivideAndConquer*, and *PipelineProcessing* (Sections 3.1, 3.2), and 3.3 respectively); and *SupportingStructures* patterns *SPMD*, *ForkJoin*, *Reduction*, and *SharedQueue* (Sections 4.1, 4.2, 4.3, and 4.4 respectively). Each of these sections 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. To make it easier to identify patterns and pattern sections, we format pattern names as *SomePattern* and pattern section names as **SomeSection**.

2 The *FindingConcurrency* Design Space

The patterns in this space (presented in [MMS00]) 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 specifically, 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.

The patterns in this design space are organized as illustrated in Figure 1. The main

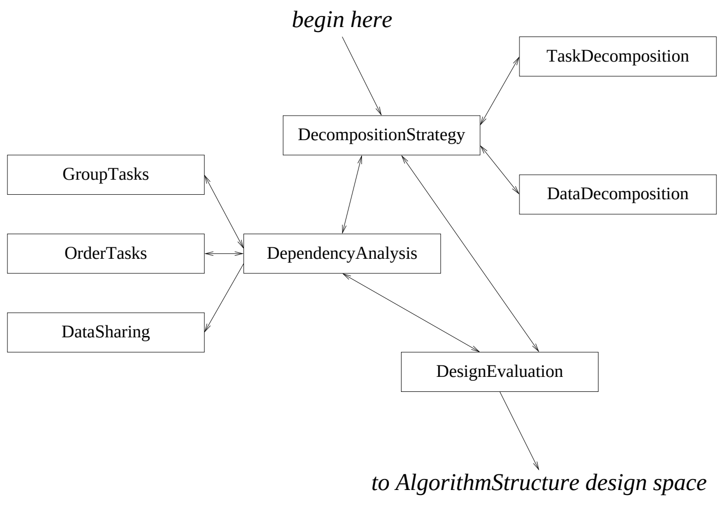


Figure 1: Organization of the *FindingConcurrency* design space.

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 identified, 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 figure 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.

Example analysis

As an example of the analysis performed by the patterns in this design space, consider the following problem taken from the field of medical imaging. (This example is presented in more detail in [MMS00].) We can decompose this problem in two ways — in terms of tasks and in terms of data.

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 difficult 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 defining a full trajectory. To create a statistically significant 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. Another approach would be to partition the body into sections and assign different sections to different processing elements.

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 flexibility 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 final 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.

3 The *AlgorithmStructure* Design Space

This design space is concerned with structuring the algorithm to take advantage of potential concurrency. That is, the designer working at this level reasons about how to use the concurrency exposed in the previous level. Patterns in this space describe overall strategies for exploiting concurrency.

Patterns in this design space can be divided into the following three groups, plus *ChooseStructure* pattern (Section 3.1 in this paper), which addresses the question of how to use the analysis performed by using the *FindingConcurrency* patterns to select an appropriate pattern from those in this space.

"Organize by ordering" patterns

These patterns are used when the *ordering of groups of tasks* is the major organizing principle for the parallel algorithm. This group has two members, reflecting two ways task groups can be ordered. One choice represents "regular" orderings that do not change during the algorithm; the other represents "irregular" orderings that are more dynamic and unpredictable.

- *PipelineProcessing* (Section 3.3 of this paper): The problem is decomposed into ordered groups of tasks connected by data dependencies.
- *AsynchronousComposition*: The problem is decomposed into groups of tasks that interact through asynchronous events.

"Organize by tasks" patterns

These patterns are those for which the tasks themselves are the best organizing principle. There are many ways to work with such "task-parallel" problems, making this the largest pattern group.

- *EmbarrassinglyParallel* (presented in [MMS99]): The problem is decomposed into a set of independent tasks. Most algorithms based on task queues and random sampling are instances of this pattern.
- *SeparableDependencies* (presented in [MMS99]): The parallelism is expressed by splitting up tasks among units of execution (threads or processes). Any dependencies between tasks can be pulled outside the concurrent execution by replicating the data prior to the concurrent execution and then combining the replicated data after the concurrent execution. This pattern applies when variables involved in data dependencies are written but not subsequently read during the concurrent execution.
- *ProtectedDependencies*: The parallelism is expressed by splitting up tasks among units of execution. In this case, however, variables involved in data dependencies are both read and written during the concurrent execution; thus, they cannot be pulled outside the concurrent execution but must be managed during the concurrent execution of the tasks.

- *DivideAndConquer* (Section 3.2 of this paper): The problem is solved by recursively dividing it into subproblems, solving each subproblem independently, and then recombining the subsolutions into a solution to the original problem.

"Organize by data" patterns

These patterns are those for which the decomposition of the data is the major organizing principle in understanding the concurrency. There are two patterns in this group, differing in how the decomposition is structured (linearly in each dimension or recursively).

- *GeometricDecomposition* (presented in [MMS99]): The problem space is decomposed into discrete subspaces; the problem is then solved by computing solutions for the subspaces, with the solution for each subspace typically requiring data from a small number of other subspaces. Many instances of this pattern can be found in scientific computing, where it is useful in parallelizing grid-based computations, for example.

- *RecursiveData*: The problem is defined in terms of following links through a recursive data structure.

3.1 The *ChooseStructure* Pattern

Problem

After you have analyzed your problem to identify exploitable concurrency, how do you use the results to choose a structure for the parallel algorithm?

Context

The first phase of designing a parallel algorithm usually consists of analyzing the problem to identify exploitable concurrency, usually by using the patterns of the *FindingConcurrency*[1] design space. After performing this analysis, you should have (1) a way of decomposing the problem into a number of tasks, (2) an understanding of how the problem's data is decomposed onto and shared among the tasks, and (3) an ordering of task groups to express temporal or other constraints among the tasks. To refine the design further and move it closer to a program that can execute these tasks concurrently, you need to map the concurrency onto the multiple units of execution (UEs)[2] that run on a parallel computer.

Of the countless ways to define an algorithm structure, most follow one of nine basic design patterns. The key issue is to decide which pattern is most appropriate for your problem.

[1]A set of patterns in our pattern language; see Section 2 of this paper and [MMS00].

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

Forces

There are competing forces to keep in mind in deciding which overall structure fits your problem best:

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

This pattern describes a way of balancing these forces and choosing an overall structure for the algorithm.

Solution

Overview

Our approach to using the results of the preliminary analysis to choose an overall structure for the algorithm has four major steps:

- **Target platform:** What constraints are placed on the parallel algorithm by the target machine or programming environment?
- **Major organizing principle:** When you consider the concurrency in your problem, is there a particular way of looking at it that stands out and provides a high-level mechanism for organizing this concurrency? Notice that in some situations, a good design may make use of multiple algorithm structures (combined hierarchically, compositionally, or in sequence), and this is the point at which to consider whether such a design makes sense for your problem.
- **The *AlgorithmStructure* decision tree:** For each subset of tasks, how do you select an *AlgorithmStructure* design pattern that most effectively defines how to map the tasks onto UEs?
- **Re-evaluation:** Is this chosen *AlgorithmStructure* pattern (or patterns) suitable for your target platform?

Steps

- **Consider the target platform.** What constraints are placed on your design by the target computer and its supporting programming environments? In an ideal world, it would not be necessary to consider such questions at this stage of the design, and doing so works against keeping the program portable (which is also desirable). This is not an ideal world, however, and if you do not consider the major features of your target platform, you risk coming up with a design that is difficult to implement efficiently.

The primary issue is how many UEs (processes or threads) your system will effectively support, since an algorithm that works well for ten UEs may not work well at all for hundreds of UEs. You do not necessarily need to decide on a specific number (in fact to do so would overly constrain the applicability of your design), but you do need to decide on an order-of-magnitude number of UEs.

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

When thinking about both of these issues — the number of UEs and the cost of sharing information — avoid the tendency to over-constrain your design. Software usually outlives hardware, so over the course of a program's life, you may need to support a tremendous range of target platforms. You want your algorithm to meet the needs of your target platform, but at the same time, you want to keep it flexible so it can adapt to different classes of hardware.

Also, remember that in addition to multiple UEs and some way to share information among them, a parallel computer has one or more programming environments that can be used to implement parallel algorithms. Do you know which parallel programming environment you will use for coding your algorithm? If so, what does this imply about how tasks are created and how information is shared among UEs?

- **Identify the major organizing principle.** Consider the concurrency you found using the patterns of the *FindingConcurrency* design space. It consists of tasks and groups of tasks, data (both shared and task-local), and ordering constraints among task groups. Your next step is to find an algorithm structure that represents how this concurrency maps onto the UEs. The first step is to find the *major organizing principle* implied by the concurrency. This usually falls into one of three camps: *organization by orderings*, *organization by tasks*, or *organization by data*. Algorithms in the first two groups are task-parallel, since the design is guided by how the computation is decomposed into tasks. Algorithms in the third group are data-parallel, because how the data is decomposed guides the algorithm. We now consider each of these in more detail.

  For some problems, the major feature of the concurrency is the presence of well-defined interacting groups of tasks, and the key issue is how these groups are ordered with respect to each other. For example, a GUI-driven program might be parallelized by decomposing it into a task that accepts user input, a task that displays output, and one or more background computational tasks, with the tasks interacting via "events" (e.g., the user does something, or a part of the background computation completes). Here the major feature of the concurrency is the way in which these distinct task groups interact.

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

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

As you think about your problem and search for the most productive way to begin organizing your concurrency and mapping it onto UEs, remember that the most effective parallel algorithm design may be hierarchical or compositional: It often happens that the very top level of the design is a sequential composition of one or more *AlgorithmStructure* patterns (for example, a loop whose body is an instance of an *AlgorithmStructure* pattern). Other designs may be organized hierarchically, with one pattern used to organize the interaction of the major task groups and other patterns used to organize tasks within the groups.

- **Identify the pattern(s) to use.** Having considered the questions raised in the preceding sections, you are now ready to select an algorithm structure, guided by an understanding of constraints imposed by your target platform, an appreciation of the role of hierarchy and composition, and a major organizing principle for your problem. You make the selection by working through the decision tree presented in Figure 2 Starting at the top of the tree, consider your concurrency and the major organizing principle, and use this information to select one of the three branches of the tree; then follow the discussion below for the appropriate subtree. Notice again that for some problems the final design may combine more than one algorithm structure; if no one of these structures seems suitable for your problem, it may be necessary to divide the tasks making up your problem into two or more groups, work through this procedure separately for each group, and then determine how to combine the resulting algorithm structures.

  – *Organize By Ordering.* Select the *OrganizeByOrdering* subtree when the major organizing principle is how the groups of tasks are ordered with respect to each other. This pattern group has two members, reflecting two ways task groups can be ordered. One choice represents "regular" orderings that do not change during the algorithm; the other represents "irregular" orderings that are more dynamic and unpredictable.

    ∗ *PipelineProcessing*[3]: The problem is decomposed into ordered groups of tasks connected by data dependencies.

[3] Section 3.3 of this paper.

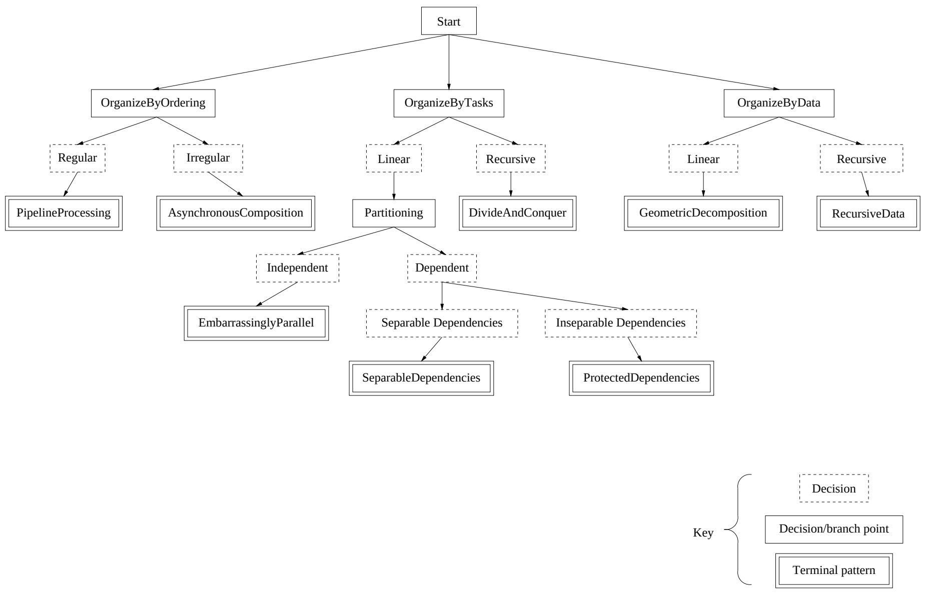


Figure 2: Decision tree for the *AlgorithmStructure* design space.

  * *AsynchronousComposition*[4]: The problem is decomposed into groups of tasks that interact through asynchronous events.

- *Organize By Tasks.* Select the *OrganizeByTasks* subtree when the execution of the tasks themselves is the best organizing principle. There are many ways to work with such "task-parallel" problems, making this the largest group of patterns. The first decision to make is how the tasks are enumerated. If they can be gathered into a set linear in any number of dimensions, take the *Partitioning* branch. If the tasks are enumerated by a recursive procedure, take the *Tree* branch.

  If the *Partitioning* branch is selected, the next question is to consider the dependencies among the tasks. If there are no dependencies among the tasks (i.e., the tasks do not need to exchange information), then you can use the following pattern for your algorithm structure:

  * *EmbarrassinglyParallel*[5]: The problem is decomposed into a set of independent tasks. Most algorithms based on task queues and random sampling are instances of this pattern.

  If there are dependencies among the tasks, you need to decide how they can be resolved. For a large class of problems, the dependencies are expressed by write-once updates or associative accumulation into shared data struc-

[4]A pattern in the *AlgorithmStructure* design space.

[5]A pattern in the *AlgorithmStructure* design space; see [MMS99].

tures. In these cases, the dependencies are separable and you can use the following algorithm structure:

- *SeparableDependencies*[6]: The parallelism is expressed by splitting up tasks among units of execution. Any dependencies among tasks can be pulled outside the concurrent execution by replicating the data prior to the concurrent execution and then reducing the replicated data after the concurrent execution. That is, once the data has been replicated, the problem is greatly simplified and looks similar to the embarrassingly parallel case.

If the dependencies involve true information-sharing among concurrent tasks, however, you cannot use a trick to make the concurrency look like the simple embarrassingly parallel case. There is no way to get around explicitly managing the shared information among tasks, and you will probably need to use the following pattern:

- *ProtectedDependencies*[7]: The parallelism is expressed by splitting up tasks among units of execution. In this case, however, variables involved in data dependencies are both read and written during the concurrent execution and thus cannot be pulled outside the concurrent execution but must be managed during the concurrent execution of the tasks. Notice that the exchange of messages is logically equivalent to sharing a region of memory, so this case covers more than just traditional shared-memory programs.

This completes the *Partitioning* branch; now consider the patterns in the *Tree* branch. Here we have two cases. These cases are very similar, differing in how the subproblems are solved once the tasks have been recursively generated:

- *DivideAndConquer*[8]: The problem is solved by recursively dividing it into subproblems, solving each subproblem independently, and then recombining the subsolutions into a solution to the original problem.

– *Organize By Data.* Select the *OrganizeByData* subtree when the decomposition of the data is the major organizing principle in understanding the concurrency. There are two patterns in this group, differing in how the decomposition is structured — linearly in each dimension or recursively.

- *GeometricDecomposition*[9]: The problem space is decomposed into discrete subspaces; the problem is then solved by computing solutions for the subspaces, with the solution for each subspace typically requiring data from a small number of other subspaces. Many instances of this pattern can be found in scientific computing, where it is useful in parallelizing grid-based computations, for example.
- *RecursiveData*[10]: The problem is defined in terms of following links

[6]A pattern in the *AlgorithmStructure* design space; see [MMS99].
[7]A pattern in the *AlgorithmStructure* design space.
[8]Section 3.2 of this paper.
[9]A pattern in the *AlgorithmStructure* design space; see [MMS99].
[10]A pattern in the *AlgorithmStructure* design space.

through a recursive data structure.

- **Re-evaluate.** After choosing one or more *AlgorithmStructure* patterns to be used in your design, skim through their descriptions to be sure they are reasonable suitable for your target platform. (For example, if your target platform consists of a large number of workstations connected by a slow network, and one of your chosen AlgorithmStructure patterns requires frequent communication between tasks, you are likely to have trouble implementing your design efficiently.) If the chosen patterns seem wildly unsuitable for your target platform, try identifying a secondary organizing principle and working through the preceding step again.

Examples

Medical imaging

For example, consider the medical imaging problem described in *DecompositionStrategy*[11]. This application simulates a large number of gamma rays as they move through a body and out to a camera. One way to describe the concurrency is to define the simulation of each ray as a task. Since they are all logically equivalent, we put them into a single task group. The only data shared among the tasks are read-only accesses to a large data structure representing the body. Hence the tasks do not depend on each other.

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

Thus, we should be able to choose a suitable structure by working through the decision tree in Figure 2. Given that in this problem the tasks are independent, the only issue we really need to worry about as we select an algorithm structure is how to map these tasks onto UEs. That is, for this problem the major organizing principle seems to be the way the tasks are organized, so we start by following the *OrganizeByTasks* branch.

We now consider the nature of our set of tasks. Are the tasks arranged hierarchically, or do they reside in an unstructured or flat set? For this problem, the tasks are in an unstructured set with no obvious hierarchical structure among them, so we follow the *Partitioning* branch of the decision tree.

The next decision takes into account the dependencies among the tasks. In this example, the tasks are independent. This implies that the algorithm structure to use for this problem is described in *EmbarrassinglyParallel*. (Notice, by the way, that an embarrassingly parallel algorithm should not be viewed as trivial to implement — much careful design work is still needed to come up with a correct program that efficiently schedules the set of independent tasks for execution on the UEs.)

[11] A pattern in the *FindingConcurrency* design space; see [MMS00]. This example is also summarized in Section 2 of this paper.

Finally, we review this decision in light of possible target-platform considerations. As we observed earlier, the key features of this problem (the large number of tasks and their independence) make it unlikely that we will need to reconsider because the chosen structure will be difficult to implement on the target platform. Nevertheless, to be careful we also review *EmbarrassinglyParallel*; fortunately, it appears to be suitable for a variety of target platforms.

3.2 The *DivideAndConquer* pattern

Problem

How can you exploit the potential concurrency in a problem that can be solved using the divide-and-conquer strategy?

Context

Consider the divide-and-conquer strategy employed in many sequential algorithms. With this strategy, a problem is solved by splitting it into subproblems, solving them independently, and merging their solutions into a solution for the whole problem. The subproblems can be solved directly, or they can in turn be solved using the same divide-and-conquer strategy, leading to an overall recursive program structure. The potential concurrency in this strategy is not hard to see: Since the subproblems are solved independently, their solutions can be computed concurrently. Figure 3 illustrates the strategy and the potential concurrency.

The divide-and-conquer strategy can be more formally described in terms of the following functions (where *N* is a constant):

- `Solution solve(Problem P)`: Solve a problem (returns its solution).
- `Problem[] split(Problem P)`: Split a problem into *N* subproblems, each strictly smaller than the original problem (returns the subproblems).
- `Solution merge(Solution[] subS)`: Merge *N* subsolutions into solution (returns the merged solution).
- `boolean baseCase(Problem P)`: Decide whether a problem is a "base case" that can be solved without further splitting (returns `true` if base case, `false` if not).
- `Solution baseSolve(Problem P)`: Solve a base-case problem (returns its solution).

The strategy then leads to the top-level program structure shown in Figure 4.

Indications

Use *DivideAndConquer* when:

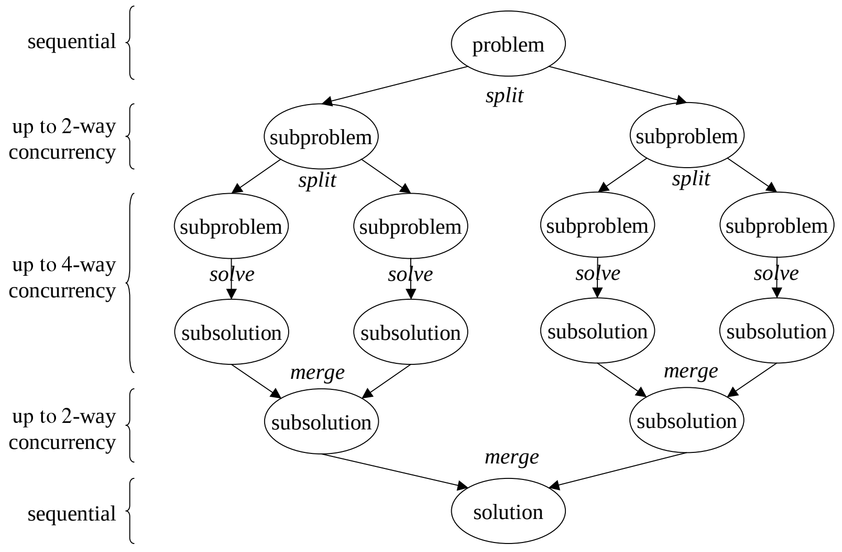


Figure 3: The divide-and-conquer strategy.

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

Figure 4: Sequential pseudocode for the divide-and-conquer strategy.

- The problem can be solved using the divide-and-conquer strategy, with subproblems being solved independently.

This pattern is particularly effective when:

- The amount of work required to solve the base case is large compared to the amount of work required for the recursive splits and merges.

- The split produces subproblems of roughly equal size.

Forces

- The traditional divide-and-conquer strategy is a widely useful approach to algorithm design. Algorithms based on this strategy are almost trivial to parallelize based on the obvious exploitable concurrency.
- As Figure 3 suggests, however, the amount of exploitable concurrency varies over the life of the program; at the outermost level of the recursion (initial split and final merge) there is no exploitable concurrency, while at the innermost level (base-case solves) the number of concurrently-executable tasks is the number of base-case problems (which is often the same as the problem size). Ideally, you would like to always have at least as many concurrently-executable tasks as processors, and clearly this pattern falls short in that respect, and the problem only gets worse as you increase the number of processors, so in general this pattern does not scale well.
- This pattern is more efficient when the subproblems into which each problem is split are roughly equal in size / computational complexity.

Solution

Overview

If the subproblems of a given problem can be solved independently, then you can solve them in any order you like, including concurrently. This means that you can produce a parallel application by replacing the `for` loop of Figure 4 with a parallel-for construct, so that the subproblems will be solved concurrently rather than in sequence. It is worth noting at this point that program correctness is independent of whether the subproblems are solved sequentially or concurrently, so you can even design a hybrid program that sometimes solves them sequentially and sometimes concurrently, based on which approach is likely to be more efficient (more about this later).

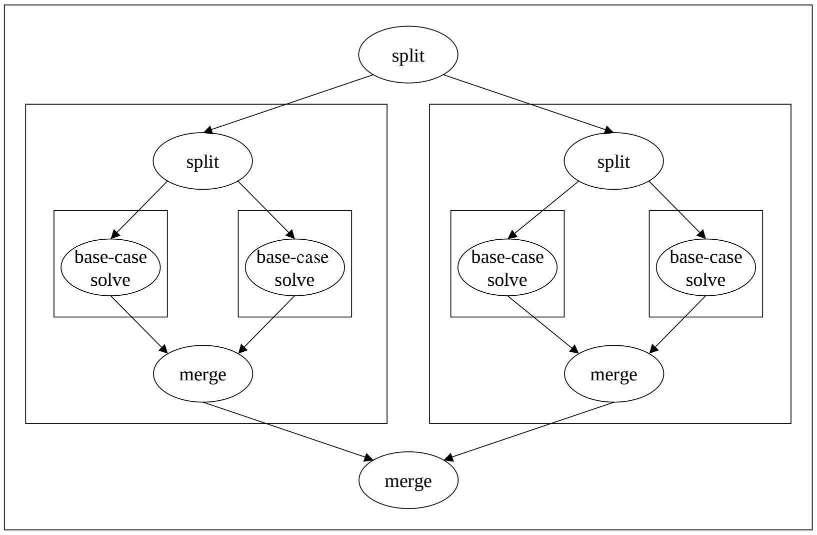
This strategy can be mapped onto a design in terms of tasks by defining one task for each invocation of the `solve` function, as illustrated in Figure 5 (rectangular boxes correspond to tasks).

Note the recursive nature of the design, with each task in effect generating and then absorbing a subtask for each subproblem.

Note also that either the split or the merge phase can be essentially absent:

- No split phase is needed if all the base-case problems can be derived directly from the whole problem (without recursive splitting). In this case, the overall design will look like the bottom half of Figures 3 and 5.
- No merge phase is needed if the problem can be considered solve when all of the base-case problems have been identified and solved. In this case, the overall design will look like the top half of Figures 3 and 5.

Designs based on this pattern include the following key elements:



(boxes correspond to tasks)

Figure 5: Parallelizing the divide-and-conquer strategy.

- Definitions of the functions described in the **Context** section above (`solve`, `split`, `merge`, `baseCase`, and `baseSolve`).
- A way of scheduling the tasks that efficiently exploits the available concurrency (subproblems can be solved concurrently).

Key elements

- **Definitions of functions.** It is usually straightforward to produce a program structure that defines the required functions: What is required is almost the same as the equivalent sequential program, except for code to schedule tasks, as described in the next section.
- **Scheduling the tasks.** Where a parallel divide-and-conquer program differs from its sequential counterpart is that the parallel version is also responsible for scheduling the tasks in a way that exploits the potential concurrency (subproblems can be solved concurrently) efficiently.

  The simplest approach is to simply replace the sequential `for` loop over subproblems with a parallel-`for` construct, allowing the corresponding tasks to execute concurrently. (Thus, in Figure 5, the two lower-level splits execute concurrently, the four base-case solves execute concurrently, and the two lower-level merges execute concurrently.) To improve efficiency (as discussed later in this section), you can also use a combination of parallel-`for` constructs and sequential `for` loops, typically using parallel-`for` at the top levels of the recursion and sequential `for` at the more deeply nested levels. In effect, this approach combines parallel divide-and-conquer with sequential divide-and-conquer.

Correctness issues

Most of the correctness issues in implementing this pattern are the same ones involved in sequential divide-and-conquer, plus a few additional restrictions to make the concurrency work properly.

- Considering first the sequential divide-and-conquer strategy expressed in the pseudocode of Figure 4, you can guarantee that `solve(P)` returns a correct solution of `P` if the other functions meet the following specifications, expressed in terms of preconditions and postconditions. (As before, *N* is an integer constant.)
  - `Solution baseSolve(Problem P)`:

    Precondition: `baseCase(P) = true`.

    Postcondition: returned value is a solution of `P`.
  - `Problem[] split(Problem P)`:

    Precondition: `baseCase(P) = false`.

    Postcondition: returned value is an array of *N* subproblems, each strictly smaller than `P`, whose solutions can be combined to give a solution of `P`. Here, "strictly smaller" means smaller with respect to some integer measure that, when small enough, indicates a base-case problem. (This ensures that the recursion is finite.)
  - `Solution merge(Solution[] subS)`:

    Precondition: `subS` is an array of *N* solutions such that for some problem `P` and array of subproblems `subP = split(P)`, `subS[i]` is a solution of `subP[i]`, for all *i* from 1 through *N*.

    Postcondition: returned value is a solution of `P`.
- To make the concurrency work, it is sufficient for the solutions of subproblems to be computed independently. That is, for two distinct subproblems `subP[i]` and `subP[j]` of `P`, `solve(subP[i])` and `solve(subP[j])` must be computed independently. This will be true if neither call to `solve` modifies variables shared with the other call. If the solutions of subproblems cannot be computed independently, then any access to shared variables must be protected with appropriate synchronization. This, of course, tends to reduce the efficiency of the calculation.

Efficiency issues

Effective use of this pattern depends on reducing the fraction of the program's lifespan during which there are fewer concurrently-executable tasks than processors, and there are several factors that contribute to this goal:

- Having a problem whose split and merge operations are computationally trivial compared to its base-case solve.
- Having a problem size that is large compared to the maximum number of processors available on the target environment.
- Reducing the number of levels of recursion required to arrive at the base-case solve by splitting each problem into more subproblems. This generally requires some algorithmic cleverness but can be quite effective, especially in the limiting case of "one-deep divide-and-conquer", in which the initial split is into P subproblems, where P is the number of available processors. See the **Related Patterns** section for more discussion of this strategy.
- If problem size is large compared to the number of available processors, at some point in the computation the number of concurrently-executable tasks will exceed the number of processors. If you take the simple approach of always using the parallel-`for` construct to schedule the tasks corresponding to subproblems, this approach produces a situation in which at some point the number of units of execution exceeds the number of available processors. If such a situation would be inefficient in the target environment (i.e., if context-switching among UEs[12] is expensive), or if there is significant overhead associated with the parallel-`for` construct, it will probably be more efficient to use the parallel-`for` construct only for the outer levels of the recursion, switching to a sequential loop when the total number of subproblems (number of subproblems per split multiplied by recursion level) exceeds number of available processors.

Examples

Mergesort

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

- The base case is an array of size 1, which is already sorted and can thus be returned without further processing.
- In the split phase, the array is split by simply partitioning it into two contiguous subarrays, each of size $N/2$ (or $(N+1)/2$ and $(N-1)/2$, if N is odd).
- In the solve-subproblems phase, the two subarrays are sorted (by applying the mergesort procedure recursively).

[12]Units of execution — generic term for a collection of concurrently-executing entities, usually either processes or threads.

- In the merge phase, the two (sorted) subarrays are recombined into a single sorted array in the obvious way.

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

Matrix diagonalization

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

- The base case is a 1-by-1 matrix, which is already diagonal and can be returned without further processing.
- The split phase consists of finding matrix T' and vectors u, v, such that $T = T' + uv^T$, and T' has the form

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

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

Details can be found in [DS87] or in [GL89].

Other known uses

- Classical graph and other algorithms. Any introductory algorithms text will have many examples of algorithms based on the divide-and-conquer strategy, most of which can be parallelized with this pattern. (As noted in the **Consequences** section, however, such parallelizations are not always efficient.)
- Leslie Grignard's Fast Multipole Algorithm.
- Floating Point Systems' FFT (Fast Fourier Transform — an algorithm for computing discrete Fourier transforms).
- Tree-based reductions, particularly for the PRAM model, as described in [J9́2].
- Certain well-known algorithms for solving the N-body problem, for example the Barnes-Hut algorithm and some algorithms of John Salmon.

Related Patterns

It is interesting to note that just because an algorithm is based on a (sequential) divide-and-conquer strategy does not mean that it must be parallelized with *DivideAndConquer*. A hallmark of this pattern is the recursive arrangement of the tasks, leading to a varying amount of concurrency. Since this can be inefficient, it is often better to rethink the problem such that it can be mapped onto some other pattern, such as *GeometricDecomposition* or *SeparableDependencies*[13].

3.3 The *PipelineProcessing Pattern*

Problem

If your problem can be solved by an algorithm in which data flows through a sequence of tasks or stages (a *pipeline*), how can you exploit the potential concurrency in this approach?

Context

The basic idea of this pattern is much like the idea of an assembly line: To perform a sequence of essentially identical calculations, each of which can be broken down into the same sequence of steps, we set up a "pipeline", one stage for each step, with all stages potentially executing concurrently. Each of the sequence of calculations is performed by having the first stage of the pipeline perform the first step, and then the second stage the second step, and so on. As each stage completes a step of a calculation, it passes the calculation-in-progress to the next stage and begins work on the next calculation.

This may be easiest to understand by thinking in terms of the assembly-line analogy: For example, suppose the goal is to manufacture a number of cars, where the manufacture of each car can be separated into a sequence of smaller operations (e.g., installing a windshield). Then we can set up an assembly line (pipeline), with each operation assigned to a different worker. As the car-to-be moves down the assembly line, it is built up by performing the sequence of operations; each worker, however, performs the same operation over and over on a succession of cars.

Returning to a more abstract view, if we call the calculations to be performed C_1, C_2, and so forth, then we can describe operation of a *PipelineProcessing* program thus: Initially, the first stage of the pipeline is performing the first operation of C_1. When that completes, the second stage of the pipeline performs the second operation on C_1; simultaneously, the first stage of the pipeline performs the first stage of C_2. When both complete, the third stage of the pipeline performs the third operation on C_1, the second stage performs the second operation on C_2, and the first stage performs the first operation on C_3. Figure 6 illustrates how this works for a pipeline consisting of four stages.

This idea can be extended to include situations in which some operations can be performed concurrently. Figure 7 illustrates two pipelines, each with four stages. In

[13] Patterns in the *AlgorithmStructure* design space; see [MMS99].

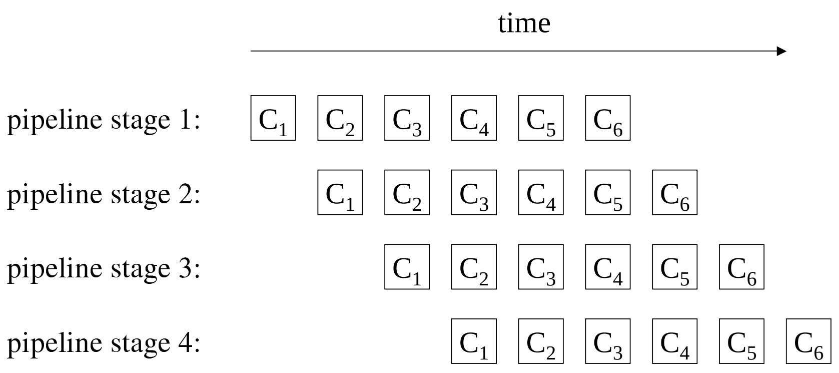


Figure 6: Pipeline stages.

the second pipeline, the third stage consists of two operations that can be performed concurrently.

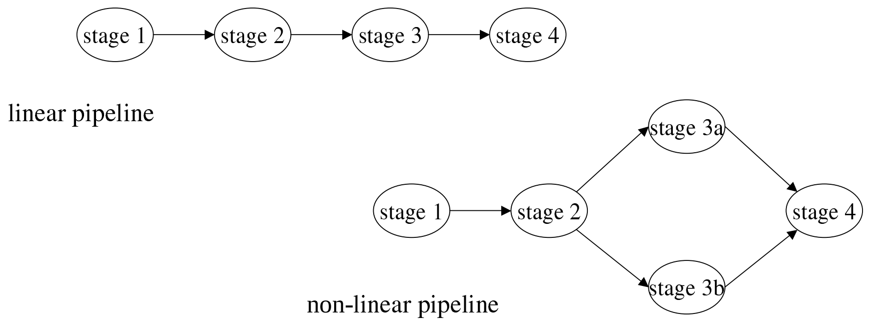


Figure 7: Example pipelines.

These figures suggest that we can represent a pipeline as a directed graph, with vertices corresponding to elements of the calculation and edges indicating dataflow. To preserve the idea of the pipeline, it is necessary that this graph be acyclic, and it is probably best if it does not depart too much from a basically linear structure, in which the elements can be divided into stages, with each stage communicating only with the previous and next stages.

We can describe the linear case more formally, as follows: This pattern describes computations whose goal is to take a sequence of inputs in_1, in_2, etc. and compute a sequence of outputs out_1, out_2, etc., where the following is true:

- out_i can be computed from in_i as a composition of N functions $f^{(1)}$, $f^{(2)}$, and so on, where (letting $\circ$ denote function composition)

  $$out_i = f^{(N)} \circ \cdot \circ f^{(2)} \circ f^{(1)}(in_i)$$

- For i and j different, and m and n different, the computation of $out_i^{(m)}$ is independent of the computation $out_j^{(n)}$, where we have defined input and output sequences for the functions $f^{(k)}$ as follows:
  - $out_i^{(m)} = f^{(m)}(in_i^{(m)})$
  - $in_i^{(1)} = in_i$
  - $out_i^{(N)} = out_i$
  - $in_i^{(m+1)} = out_i^{(m)}$, for m between 1 and $N-1$.

To restate this less formally: For different stages m and n of the pipeline, and different elements C_i and C_j of the sequence of calculations to be performed, stage m of the calculation for C_i can be done independently of stage n of the calculation for C_j. This is the key restriction in this pattern and is what makes the concurrency possible.

Indications

Use *PipelineProcessing* when:

- The problem consists of performing a sequence of calculations, each of which can be broken down into distinct stages, on a sequence of inputs, such that for each input the calculations must be done in order, but it is possible to overlap computation of different stages for different inputs as indicated in Figures 6 and 7.

The pattern is particularly effective when:

- The number of calculations is large compared to the number of stages.
- It is possible to dedicate a processor to each element, or at least each stage, of the pipeline.

This pattern can also be effective in combination with other patterns:

- As part of a hierarchical design, in which the tasks making up each stage of the pipeline are internally organized using another of the *AlgorithmStructure* patterns.

Forces

- This pattern can be straightforward to implement (as described in the **Solution** section below), particularly for message-passing platforms.
- However, in a pipeline algorithm, concurrency is limited until all the stages are occupied with useful work. This is referred to as "filling the pipeline". At the tail end of the computation, again there is limited concurrency as the final item works its way through the pipeline. This is called "draining the pipeline". In order for pipeline algorithms to be effective, the time spent filling or draining the pipeline must be small compared to the total time of the computation. This pattern therefore is most effective when the number of calculations is large compared to the number of operations/stages required for each one.
- Also, either the stages of the pipeline must be kept synchronized or there must be some way of buffering work between successive stages. The pattern therefore usually works better if the operations performed by the various stages of the pipeline are all about equally computationally intensive. If the stages in the pipeline vary widely in computational effort. The slowest stage defines a bottleneck for the algorithm's aggregate throughput. Furthermore, a much slower stage in the middle of the pipeline will cause data items to back up on the input queue, potentially leading to buffer overflow problems.

Solution

Overview

Viewing the pattern in terms of tasks, define one task for each element of the pipeline (one element per stage in a linear pipeline, possibly more for a nonlinear pipeline). Each task can be thought of as having a predecessor (in the previous stage) and a successor (in the next stage), with obvious exceptions for tasks corresponding to the first and last stages of the pipeline and a straightforward generalization to nonlinear pipelines (where a task can have multiple predecessors or successors). Data dependencies are defined as follows. Each task requires as input a sequence of input items from its predecessor (or synchronized sequences from its predecessors); for each input item it generates an output item to be used by its successor (or synchronized groups of items, each group to be distributed among its successors).

Designs based on this pattern include the following key elements:

- A way of defining the elements of the pipeline, where each element corresponds to one of the functions that make up the computation. In a linear pipeline (such as the "linear pipeline" of Figure 7), these elements are the stages of the pipeline; in a more complex pipeline (such as the "non-linear pipeline" of Figure 7) there can be more than one element per pipeline stage. Each pipeline element will correspond to one task.
- A way of representing the dataflow among pipeline elements, i.e., how the functions are composed.

- A way of scheduling the tasks.

Key elements

- **Defining the elements of the pipeline.** What is needed here is a program structure to hold the computations associated with each stage. There are many ways to do this. A particularly effective approach is an SPMD[14] program in which the ID associated with the UE[15] selects options in a case statement, with each case corresponding to a stage of the pipeline.

- **Representing the dataflow among pipeline elements.** What is needed here is a mechanism that provides for the orderly flow of data between stages in the pipeline. This is relatively straightforward to accomplish in a message-passing environment by assigning one process to each function and implementing each function-to-function connection (between successive stages of the pipeline) via a sequence of messages between the corresponding tasks. Since the stages are hardly ever perfectly synchronized, and the amount of work carried out at different stages almost always varies, this flow of data between pipeline stages must usually be both buffered and ordered. Most message-passing environment (e.g., MPI) make this easy to do.

  If a message-passing programming environment is not a good fit with the target platform, you will need to explicitly connect the stages in the pipeline with a buffered channel. Such a buffered channel can implemented as a shared queue (using *SharedQueue*[16]).

- **Scheduling the tasks.** What is needed here is a way of scheduling the tasks that make up the design. Usually all tasks are scheduled to execute concurrently (e.g., as elements of an SPMD program), since this avoids bottlenecks and potential deadlock.

Correctness issues

- What makes concurrency possible here is the requirement that for different stages m and n of the pipeline, and different elements C_i and C_j of the sequence of calculations to be performed, stage m of the calculation for C_i can be done independently of stage n of the calculation for C_j. The tasks that make up the pipeline should therefore be independent, except for the interaction needed to pass data from one stage to the next. This happens naturally in a distributed-memory environment; in a shared-memory environment, it can be guaranteed by (i) making sure the mechanism used to pass data from one pipeline stage to the next is correctly implemented (e.g., by using a concurrency-safe shared data

[14]"Single program, multiple data" — a parallel-programming style in which each thread or process runs the same program, but on different data.

[15]Unit of execution — generic term for one of a collection of concurrently-executing entities, usually either processes or threads.

[16]Section 4.4 of this paper.

structure such as *SharedQueue*) and (ii) not allowing tasks to modify any shared variables except those used in this mechanism.

Examples

Fourier-transform computations

A type of calculation widely used in signal processing involves performing the following computation repeatedly on different sets of data:

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

Examples of such calculations include convolution, correlation, and filtering operations, as discussed in [PFTV88].

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

- The first stage of the pipeline performs the initial Fourier transform; it repeatedly obtains one set of input data, performs the transform, and passes the result to the second stage of the pipeline.
- The second stage of the pipeline performs the desired elementwise manipulation; it repeatedly obtains a partial result (of applying the initial Fourier transform to an input set of data) from the first stage of the pipeline, performs its manipulation, and passes the result to the third stage of the pipeline.
- The third stage of the pipeline performs the final inverse Fourier transform; it repeatedly obtains a partial result (of applying the initial Fourier transform and then the elementwise manipulation to an input set of data) from the second stage of the pipeline, performs the inverse Fourier transform, and outputs the result.

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

Known uses

- Image processing applications.
- Problems in the CMU task-parallel Fortran test suite [cmu].
- Other pipelined computations, as described in [J́92].

Related Patterns

This pattern is very similar to the *Pipes and Filters* pattern of [BMR$^+$96]; the key difference is that this pattern explicitly discusses concurrency. This pattern is similar to *AsynchronousComposition*[17] in that both patterns apply to problems where it is natural to decompose the computation into a collection of semi-independent entities. The difference is that in *PipelineProcessing*, these entities interact in a more regular way, with all "stages" of the pipeline proceeding in a loosely synchronous way, whereas in the *AsynchronousComposition* pattern, there is no such requirement, and the entities can interact in very irregular and asynchronous ways.

4 The *SupportingStructures* Design Space

The patterns of the *AlgorithmStructure* design space capture recurring solutions to the problem of turning problems into parallel algorithms. But these patterns in turn contain recurring solutions to the problem of mapping high-level parallel algorithms into programs using a particular parallel language or library. Those solutions are what the patterns in the *SupportingStructures* design space capture. Patterns in this space fall into three main groups: patterns for structuring concurrent execution, patterns that represent commonly-occurring computations, and patterns that represent commonly-occurring data structures.

Patterns for structuring concurrent execution

Patterns in this group describe ways of structuring concurrent execution, with particular attention to whether processes or threads are created statically or dynamically and whether concurrently-executing processes or threads all perform the same work. These patterns include the following:

- *SPMD* (single program, multiple data; Section 4.1 of this paper): The computation consists of *N* processes or threads running concurrently. All *N* processes or threads execute the same program code, but each operates on its own set of data. A key feature of the program code is a parameter that differentiates among the copies.

- *ForkJoin* (Section 4.2 of this paper): A main process or thread forks off some number of other processes or threads that then continue in parallel to accomplish some portion of the overall work before rejoining the main process or thread.

Patterns representing computational structures

Patterns in this group describe commonly-used computational structures; they include the following:

[17] Another pattern in the *AlgorithmStructure* design space.

- *Reduction* (Section 4.3 of this paper): A number of concurrently-executing processes or threads cooperate to perform a reduction operation, in which a collection of data items is reduced to a single item by repeatedly combining them pairwise with a binary operator.
- *MasterWorker*: A master process or thread sets up a pool of worker processes or threads and a task queue. The workers execute concurrently, with each worker repeatedly removing a task from the task queue and processing it, until all tasks have been processed or some other termination condition has been reached. In some implementations no explicit master is present.

Patterns representing data structures

Patterns in this group describe commonly-used shared data structures; they include the following:

- *SharedQueue* (Section 4.4 of this paper): This pattern represents a "thread-safe" implementation of the familiar queue abstract data type (ADT), that is, an implementation of the queue ADT that maintains the correct semantics even when used by concurrently-executing processes or threads.
- *SharedCounter*: This pattern, like the previous one, represents a "thread-safe" implementation of a familiar abstract data type, in this case a counter with an integer value and increment and decrement operations.
- *DistributedArray*: This pattern represents a class of data structures often found in parallel scientific computing, namely arrays of one or more dimensions that have been decomposed into subarrays and distributed among processes or threads.

4.1 The *SPMD* Pattern

Problem

When is it appropriate to use an "SPMD" (single program, multiple data) program structure, and what are some considerations in implementing it?

Context

It is fairly common to structure parallel algorithms in terms of a number of units of execution that are identical with respect to code but different with respect to data — i.e., according to the "SPMD" (single program, multiple data) paradigm. Each UE typically has a unique identifier that makes it possible to differentiate among them. Programming environments in which this paradigm is well-supported and widely used include MPI and PVM.

Example

As an example, consider the problem of computing the variance of a set of N numbers $a_1, \ldots, a_N$ using P processing elements. The desired quantity is given by the formula

$$\frac{(a_1 - avg)^2 + (a_2 - avg)^2 + \cdots + (a_N - avg)^2}{N}$$

where avg is the average of $a_1, \ldots, a_N$. One way to perform this computation is as a sequence of two phases, one to find the average of $a_1, \ldots, a_N$ and one to compute their variance using this average, using P UEs (one per processing element). In the first phase, the UEs first compute the sum of the elements, as described in *Reduction*,[18] and then each UE divides this sum by N to obtain the average. The second phase is similar; the UEs use the average to compute the variance using a similar strategy (compute sum of squares using *Reduction*, then divide by N). In both uses of *Reduction*, some (if not most) of the concurrency comes from partitioning the N numbers into P subsets assigning one subset to each UE, and having each UE compute a partial sum for its subset. (If the numbers are stored in an array, *DistributedArray*[19] describes how to perform such a partitioning.)

Performing this calculation using *SPMD* requires P UEs (created once at the start of the computation), each performing the following calculation: Sum the elements in the appropriate subset, communicate/synchronize with other UEs to compute the global sum (as described in *Reduction*), and divide the result by N. Perform phase two in a similar way, using the same P UEs used in the first phase.

In this example, each UE is performing essentially the same calculations in the same order (although the details of the reduction operation might be different for different UEs), but each is operating on a different set of data ("its" subset of the input numbers).

Forces

Forces to consider when deciding whether to use this pattern include the following:

- Can the computation be structured as a set of tasks that all start up when the program begins, persist throughout its execution, and execute essentially the same computation?

- Is it feasible to have all UEs execute the same code (or essentially the same code — details can depend on data that varies among UEs)? Parallelization strategies based on data decomposition are likely candidates for such a program structure.

- Does the target programming environment require an SPMD program structure (e.g., some implementations of MPI)? If not, is the cost of creating UEs sufficiently high to make this program structure more attractive than alternatives requiring dynamic creation and destruction of UEs, such as *ForkJoin*[20]?

[18] Section 4.3 of this paper.
[19] Another pattern in the *SupportingStructures* design space.
[20] Section 4.2 of this paper.

Solution

Overview

This pattern forms the top-level structure of a parallel program. In a program based on *SPMD*, many copies of an essentially sequential program are started simultaneously and execute concurrently, using message-passing or other mechanisms to communicate and synchronize. Each copy has its own set of program variables, usually including a unique identifier that allows different UEs to perform different work if necessary. For example, many computations that can be parallelized using *GeometricDecomposition*[21] require that calculations for boundary points be slightly different from calculations for interior points, so a UE whose data includes boundary points will perform calculations slightly different from a UE whose data consists only of interior points. The computation as a whole is finished when all UEs terminate. Figure 8 illustrates an SPMD program using message-passing.

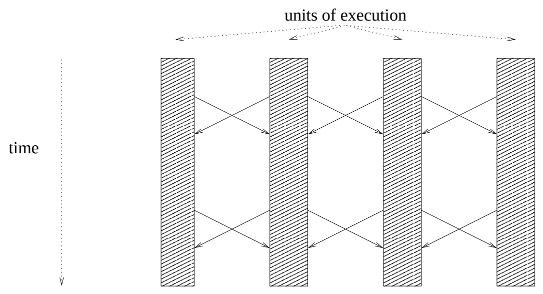


Figure 8: SPMD program using message-passing.

Applicability issues

There are many issues to be resolved in applying this pattern. Most of them depend on the context in which the pattern is applied and so are discussed more fully in the *AlgorithmStructure* patterns that suggest using *SPMD*. They include the following:

- How to partition the original problem's data into (i) common data (to be shared among UEs, via replication if necessary) and (ii) process data (data unique to a UE — typically a portion of a large data structure that is being distributed among UEs). Depending on the target platform and the problem, it may be appropriate to duplicate some process data.
- How the UEs interact — via message-passing or some other mechanism.

[21] A pattern in the *AlgorithmStructure* design space; see [MMS99].

Key elements

- **Code to execute.** This is the program or code to be executed by each UE. Notice that while all UEs execute the same program in this pattern, they may take different paths through it.

- **Common data.** If the target platform supports shared memory, data common to all UEs can be represented as shared variables. Otherwise it must be replicated, with each UE having its own copy. There are many correctness and efficiency issues related to keeping the copies consistent, again best dealt with in the context of the pattern in which *SPMD* is applied.

- **Process data.** Data unique to each UE is easier to represent, by associating with each UE its own set of program variables. Details vary among programming environments; for an environment that does not support shared memory, such variables are the only kind possible, but many shared-memory environments also provide a way to create distinct variables for each UE.

  For some problems, data for each UE may include "shadow copies" of other UEs' process data. For example, in an application based on *GeometricDecomposition*[22] each UE's process data might include a contiguous chunk of a large multidimensional array plus "shadow copies" of data that is conceptually nearby in the array but physically part of another UE's process data. *DistributedArray*[23] discusses this in more detail.

  An important process-data variable is the unique identifier that allows us to distinguish among UEs; this identifier can be omitted only if there is no need to make such distinctions.

Related Patterns

This pattern is similar to *ForkJoin*[24] in some respects, but there are significant differences. The key difference is that this pattern must represent the very topmost structural construct in a program, while *ForkJoin* may be used several times within a program; i.e., the overall program can be a sequential or other composition of more than one instance of *ForkJoin*. A simple example illustrating this difference is given in the **Context** sections of the two patterns.

This pattern is very similar to the SPMD paradigm or model described in other literature on parallel programming; a key difference is that our pattern focuses on those programs in which all UEs are executing essentially the same computation (differing perhaps in a few details), while some definitions of "SPMD" also include programs that simulate the MPMD ("multiple program, multiple data") paradigm, in which different UEs can be performing completely different calculations.

[22]A pattern in the *AlgorithmStructure* design space; see [MMS99].
[23]Another pattern in the *SupportingStructures* design space.
[24]Section 4.2 of this paper.

4.2 The *ForkJoin* Pattern

Problem

When is it appropriate to use a "fork/join" program structure, and what are some considerations in implementing it?

Context

It is fairly common to structure parallel algorithms as a sequential composition of sections, in each of which a master unit of execution starts up a collection of additional UEs ("fork") and then waits for them to complete ("join"). Typically, but not always, all the created UEs perform essentially the same computation, usually with different parameters. Often each UE has a unique identifier that makes it possible to differentiate among them. Programming environments in which this paradigm is well-supported and widely used include Java and POSIX threads.

Example

As an example, consider the problem of computing the variance of a set of N numbers $a_1, \ldots, a_N$ using P processing elements. The desired quantity is given by the formula

$$\frac{(a_1 - avg)^2 + (a_2 - avg)^2 + \cdots + (a_N - avg)^2}{N}$$

where avg is the average of $a_1, \ldots, a_N$. One way to perform this computation is as a sequence of two phases, one to find the average of $a_1, \ldots, a_N$ and one to compute their variance using this average. In the first phase, a master UE creates P new UEs that compute the sum of the elements, as described in *Reduction*,[25] and then terminate; after they terminate, the master UE divides the resulting sum by N to obtain the average. The second phase is similar; the master UE creates P new UEs that compute the required sum of squares; after the new UEs complete their work and terminate, the master UE divides the result by N to obtain the variance. In both uses of *Reduction*, some (if not most) of the concurrency comes from partitioning the N numbers into P subsets assigning one subset to each UE, and having each UE compute a partial sum for its subset. (If the numbers are stored in an array, *DistributedArray*[26] describes how to perform such a partitioning.)

Performing this calculation using *ForkJoin* requires two uses of the pattern, one for each phase, plus a small amount of sequential calculation (the two "divide by N" operations performed by the master UE). In both uses of the pattern in this example, all UEs are performing essentially the same calculations in the same order (although the details of the reduction operation might be different for different UEs), but each is operating on a different set of data ("its" subset of the input numbers). Notice that the number of UEs varies over the course of computation, but at any point there are at most P UEs able to perform useful work, which should give good use of the P processing elements presupposed by the problem.

[25] Section 4.3 of this paper.

[26] Another pattern in the *SupportingStructures* design space.

Forces

Forces to consider when deciding whether to use this pattern include the following:

- Is it advantageous to structure part or all of the computation as described above? Such a structure is especially attractive if the number of additional UEs should be different at different points in the program or should be determined at runtime.
- Is the cost of creating UEs in the target programming environment low enough to make this structure acceptably efficient and more attractive than alternatives such as *SPMD*[27]?

Solution

Overview

This pattern can be used in many contexts within a parallel program — at the top level of the program, as one element of a sequential composition, or as the body of a loop. The additional UEs to be started can all run the same code, or different UEs can run different code. If multiple UEs run the same code, each has its own set of program variables, which can include a unique identifier that allows different UEs to (for example) operate on different parts of a shared data structure. Figure 9 illustrates a program using *ForkJoin* twice, once to start four UEs running the same code and once to start two UEs running different code.

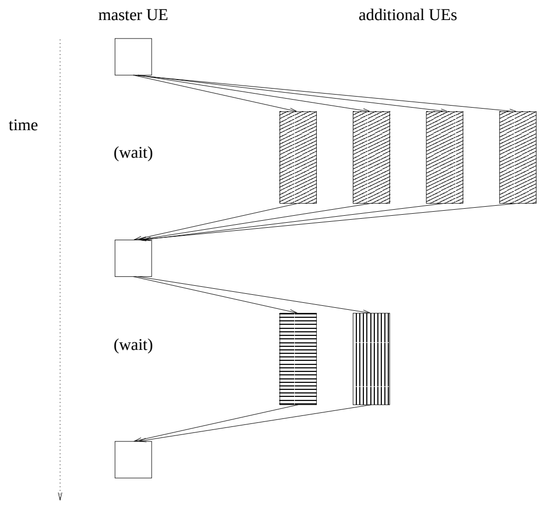


Figure 9: Program with two fork/join constructs.

[27]Section 4.1 of this paper.

Applicability issues

There are many issues to be resolved in applying this pattern. Most of them depend on the context in which the pattern is applied and so are discussed more fully in the *AlgorithmStructure* patterns that suggest using *ForkJoin*. They include the following:

- How, or whether, to partition the original problem's data into (i) common data (to be shared among UEs, via replication if necessary) and (ii) process data (data unique to a UE — typically a portion of a large data structure that is being distributed among UEs). Depending on the target platform, it may be appropriate to duplicate some process data.
- How the UEs interact, if at all — typically using locks or some other synchronization mechanism.
- Whether the master UE should continue to do other work while waiting for the additional UEs to complete their work or should be idle.

Key elements

- **Code to execute.** This is the function or functions to be executed by the additional UEs. All may execute the same code, as in *SPMD*,[28] or different UEs may execute different code.
- **Common data.** If the target platform supports shared memory, data common to all UEs can be represented as shared variables. Otherwise it must be replicated, with each UE having its own copy. There are many correctness and efficiency issues related to keeping such copies consistent, again best dealt with in the context of the pattern in which *ForkJoin* is applied.
- **Process data.** Data unique to each UE is easier to represent, by associating with each UE its own set of program variables. Details vary among programming environments; for an environment that does not support shared memory, such variables are the only kind possible, but many shared-memory environments also provide a way to create distinct variables for each UE.

  For some problems, data for each UE may include "shadow copies" of other UEs' process data. For example, in an application based on *GeometricDecomposition*[29] each UE's process data might include a contiguous chunk of a large multidimensional array plus "shadow copies" of data that is conceptually nearby in the array but physically part of another UE's process data. *DistributedArray*[30] discusses this in more detail.

  A key process-data variable is a unique identifier that allows us to distinguish among UEs; this identifier is needed if it is necessary to make such distinctions.

[28]Section 4.1 of this paper.
[29]A pattern in the *AlgorithmStructure* design space; see [MMS99].
[30]Another pattern in the *SupportingStructures* design space.

Related Patterns

This pattern is similar to *SPMD*[31] in some respects, but there are some significant differences. The key difference is that this pattern can be used repeatedly within a program, while *SPMD* must represent the very topmost structural construct in a program. A simple example illustrating this difference is given in the **Context** sections of the two patterns. *SPMD* also has significant restrictions not present in *ForkJoin*; the number of UEs is fixed for the life of the program, and all UEs must perform essentially the same computation.

Despite the almost identical names, the computational structure represented by this pattern is not the same as the "fork/join parallelism" described in [Lea00]. Lea uses this term to describe a parallelization strategy based on divide and conquer; we use it to describe a computational structure (one that is similar, however, to Lea's `FJTask.coInvoke()`).

4.3 The *Reduction* Pattern

Problem

Is there potential concurrency in a reduction operation, and if so how do you exploit it?

Context

Many parallel algorithms involve "reduction operations", in which a collection of data items is "reduced" to a single data item by repeatedly combining the data items pairwise with a binary operator, usually one that is associative and commutative. Examples of such operations include finding the sum, product, or maximum of all elements in an array. In general, we can represent such an operation as the calculation of

$$v_0 \circ v_1 \circ \cdots \circ v_{m-1}$$

where $\circ$ is a binary operator. The most general way to implement such an operation is to perform the calculation serially from left to right. This general approach is usually inefficient and scales poorly with the number of units of execution. If $\circ$ is associative, however, the above calculation contains exploitable concurrency, in that partial results over subsets of $v_0, \cdots, v_{m-1}$ can be calculated concurrently and then combined. If $\circ$ is also commutative, the calculation can be not only regrouped but also reordered, opening up additional possibilities for concurrent execution, as described later in this pattern.

Forces

There are competing forces to keep in mind in deciding whether there is exploitable concurrency in a reduction operation and if so how to exploit it:

[31]Section 4.1 in this paper.

- Is the reduction operator ($\circ$ in the formula above) associative and/or commutative? If so, data items can be combined in any order without affecting the final result, which provides exploitable concurrency. Not all reduction operators have these useful properties, however, and one question to be considered is whether the operator in question can be treated as if it were associative and/or commutative without significantly changing the result of the calculation. For example, floating-point addition is not strictly associative (because the finite precision with which numbers are represented can cause round-off errors, especially if the difference in magnitude between operands is large), but if all the data items to be added have roughly the same magnitude, it is usually close enough to associative to permit the parallelization strategies discussed below. If the data items vary considerably in magnitude, this may not be the case.

- Is it enough for a single UE to know the result of the reduction operation, or do multiple UEs need it?

This pattern discusses different ways of computing reduction operations based on an analysis of these forces.

Solution

Overview

We look at three approaches to performing reduction operations. Which should be used depends on an analysis of the forces described above.

- **Serial computation.** If the reduction operator is not associative, or cannot be treated as associative without significantly affecting the result, it will likely be necessary to perform the entire reduction serially in a single UE, as sketched in Figure 10. If only one UE needs the result of the reduction, it is probably simplest to have that UE perform the operation; if all UEs need the result, the reduction operation can be followed by a broadcast operation to communicate the result to other UEs. For simplicity, the figure shows a situation in which there are as many UEs as data items. The solution can be extended to situations in which there are more data items than UEs, but the requirement that all the actual computation be done in a single UE still holds because of the lack of associativity. (Clearly this solution completely lacks concurrency; we mention it for the sake of generality and because it might still be useful and appropriate if the reduction operation represents only a relatively small part of a computation whose other parts *do* have exploitable concurrency.)

- **Tree-based reduction.** If the reduction operator is associative or can be treated as such, then the tree-based reduction shown in Figure 11 is appropriate. This approach is particularly attractive if only one UE needs the result of the reduction, as might often be the case when this pattern is used in combination with *ForkJoin*.[32] If other UEs also need the result, the reduction operation can be

[32] Section 4.2 of this paper; the example in the **Context** section is particularly relevant.

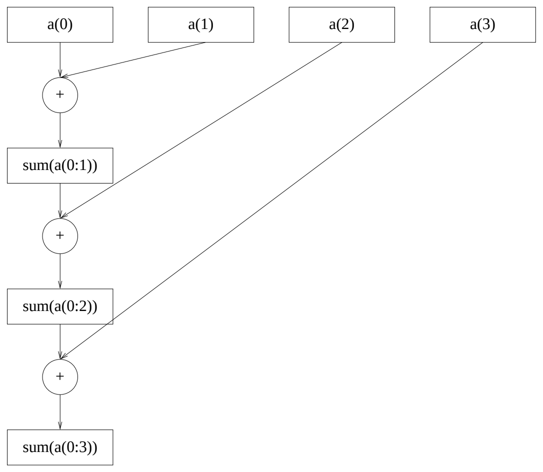


Figure 10: Serial reduction (computing the sum of `a(0)` through `a(3)`).

followed by a broadcast operation to communicate the result to other UEs. For simplicity, the figure shows a situation in which there are as many UEs as data items. The solution can be extended to situations in which there are more data items than UEs by first having each UE perform a serial reduction on a subset of the data items and then combining the results as shown. (The serial reductions, one per UE, are independent and can be done concurrently.)

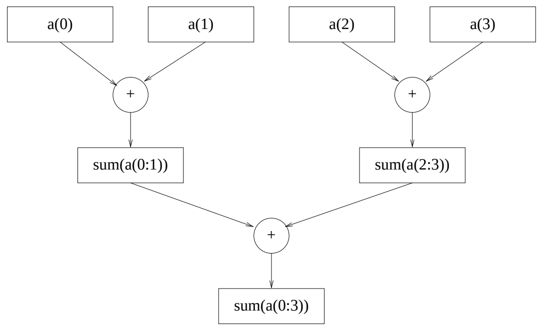


Figure 11: Tree-based reduction (computing the sum of `a(0)` through `a(3)`).

- **Recursive doubling.** If the reduction operator is associative or can be treated as such, and all UEs must know the result of the operation, then the recursive-doubling scheme of Figure 12 is likely to be appropriate,[33] as might often be the

[33] A discussion of recursive doubling can be found in [VdV94]. A similar strategy is frequently employed

case when this pattern is used in combination with *SPMD*.[34] Notice that if the reduction operator is also commutative, this strategy could be modified to first combine `a(0)` with `a(2)` and `a(1)` with `a(3)` and then combine the resulting partial sums. For simplicity, the figure shows a situation in which there are as many UEs as data items, but it can be extended to situations in which there are more data items than UEs in the same way as the tree-based reduction.

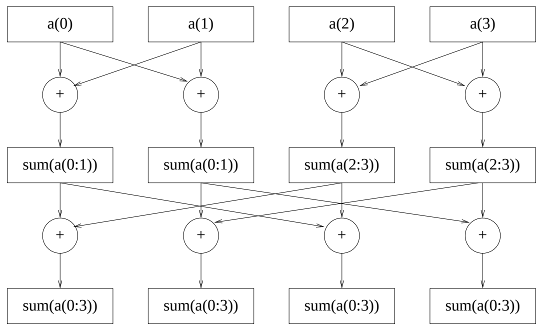


Figure 12: Recursive-doubling reduction (computing the sum of `a(0)` through `a(3)`).

Implementation considerations

This pattern is so common that many programming environments provide constructs that implement it; for example, MPI [SOHL$^+$00] provides general-purpose functions `MPI_Reduce` and `MPI_Allreduce` as well as support for several common reduction operations. If no suitable implementation exists, the following section discusses some considerations for writing one. Figures 10, 11, and 12 in effect illustrate how to represent each strategy in terms of a collection of tasks performing a “+” operation, with the arrows connecting them representing data dependencies among the tasks.

- **Serial computation.** As Figure 10 suggests, in this approach the individual combine-two-elements operations must be performed in sequence, so it is simplest to have them all performed by a single task. Because of the data dependencies (indicated by the arrows in the figure), however, some caution is required if the reduction operation is performed as part of a larger calculation involving multiple concurrent UEs; the UEs not performing the reduction can continue with other work only if they can do so without affecting the computation of the reduction operation (e.g., if multiple UEs share access to an array and one of them is computing the sum of the elements of the array, the others should not be simultaneously modifying elements of the array). In a message-passing environment, this can usually be accomplished using message-passing to enforce the

in parallel implementations of the Fast Fourier Transform (FFT) algorithm, also discussed in [VdV94].

[34] Section 4.1 of this paper; the example in the **Context** section is particularly relevant.

data-dependency constraints (i.e., the UEs not performing the reduction operation send their data to the single UE actually doing the computation). In other environments, the UEs not performing the reduction operation can be forced to wait by means of a barrier.

- **Tree-based reduction.** As Figure 11 suggests, in this approach some, but not all, of the combine-two-elements operations can be performed concurrently (e.g., in the figure we can compute `sum(a(0:1))` and `sum(a(2:3))` concurrently, but the computation of `sum(a(0:3))` must occur later). A more general sketch for performing a tree-based reduction using 2^n UEs similarly breaks down into n steps, with each step involving half as many concurrent operations as the previous step. As with the serial strategy, caution is required to make sure that the data dependencies shown in the figure are honored. In a message-passing environment, this can usually be accomplished by appropriate message-passing; in other environments, it could be implemented using barrier synchronization after each of the n steps.

- **Recursive doubling.** The situation here is very similar to that for the tree-based approach, except that all UEs are active at each step, and the data dependencies are such that a message-passing implementation will require a larger total number of messages.

Examples

Examples of reduction operations include the following:

- Computing a sum or product of elements of an array or list.

- Finding the maximum or minimum element of an array or list, or the index of the maximum or minimum element.

4.4 The *SharedQueue* Pattern

Problem

How do you implement a queue to be shared among concurrently-executing units of execution?

Context

Effective implementation of many parallel algorithms requires a queue that is to be shared among units of execution. The standard definition of a queue ADT (abstract data type) as a list supporting operations including element insertion, element removal, and predicates such as `isEmpty()`, seems sufficient from a design standpoint. Careful thought suggests, however, that a naive sequential implementation of the typical queue ADT is not guaranteed to work if multiple concurrently-executing UEs have access to an instance of the data type. What is needed in this context is an implementation of the queue ADT that can be used by concurrent callers without problems — i.e., a "thread-safe" implementation.

Forces

There are competing forces to keep in mind in deciding how to implement a queue ADT suitable for concurrent use.

- Should items be removed in the same order in which they are added (first-in, first-out), or in some priority order?
- Should the maximum size of the queue be limited (bounded), or can it grow (subject to memory limitations)?
- What should happen if a UE attempts to remove an element from an empty queue or (if the size of the queue is limited) add an element to a full queue — should the request generate an error condition, or should the requesting UE wait until the request can be satisfied?
- Will the queue be used heavily by many UEs, such that good performance is critical?

This pattern discusses different ways of implementing a shared queue based on an analysis of these forces.

Solution

Overview

Various implementation strategies are possible; which to use depends on an analysis of the forces discussed in the preceding section. We describe in this section two fairly simple strategies.

- The simplest implementation strategy is to implement the queue just as one would in a sequential program (typically using a linked list or an array) and then allow at most one UE at a time to execute any queue-related function. Enforcing this one-at-a-time access is the "mutual exclusion" problem described in textbooks on operating systems and concurrent algorithms (e.g., [BA90]) for which many solutions exist, most using synchronization mechanisms such as locks, semaphores, and monitors. This strategy is straightforward to implement and works for all types of queues (FIFO or priority, bounded or unbounded). It may not give the best performance (e.g., it does not allow concurrent additions and deletions), however, and it does not provide an obvious way to make callers wait until requests can be satisfied (e.g., until an empty queue becomes non-empty).
- Another fairly straightforward strategy is to implement the queue using a solution to the classic "producer/consumer" problem described in textbooks on operating systems and concurrent algorithms, typically using semaphores or a monitor.

This strategy explicitly provides a way to make callers wait until their requests can be satisfied, and can be adapted to work for all types of queues (FIFO or priority, bounded or unbounded — though if the queue is unbounded there is no need to include code to delay insertions if the queue is full). Depending on the details of the strategy, concurrent additions and deletions may or may not be possible; if performance is an important criterion, it may make sense to be sure the chosen approach supports this added potential concurrency.

Implementation considerations

Ideally this pattern would be implemented as part of the target programming environment. Several Java-based implementations are included in Doug Lea's `util.concurrent` package [Lea], some based on the simple strategies discussed above and some based on more complex strategies and providing additional flexibility and performance. If no suitable implementation exists, both strategies discussed previously are reasonably straightforward to implement, since the conceptually difficult parts (maintaining appropriate synchronization among UEs) are well-known problems for which many published solutions exist.

Examples

Examples of situations in which this pattern is useful include the following:

- Maintaining a queue of tasks to be performed in an application using *MasterWorker*;[35] *EmbarrassinglyParallel*[36] includes examples of problems that can be solved in this way.
- Maintaining a queue of idle workers in an application using *MasterWorker*.

Related Patterns

This pattern is similar to patterns for other shared data structures, e.g., *SharedCounter*.[37]

Acknowledgments

We thank Intel Corporation, the National Science Foundation (grant #9704697), and the Air Force Office of Scientific Research (grant #4514209-12) for their financial support. We also thank Doug Lea and Alan O'Callaghan, who acted as shepherds for our previous PLoP papers ([MMS99] and [MMS00]), and Bob Hanmer and Steve MacDonald, who shepherded the patterns in this paper. All four shepherds went beyond the call of duty in providing detailed comments and helpful insights.

[35] Another pattern in the *SupportingStructures* design space.

[36] A pattern in the *AlgorithmStructure* design space; see [MMS99].

[37] Another pattern in the *SupportingStructures* design space.

References

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Son Ltd, 1996.

[cmu] The CMU task-parallel fortran test suite. Available at `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html`.

[DS87] J.J. Dongarra and D.C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. and Stat. Comp.*, 8:S139–S154, 1987.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[J́92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Lea] Doug Lea. Overview of package `util.concurrent`. `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`.

[Lea00] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, 2000.

[MMS99] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[MMS00] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for finding concurrency for parallel application programs. In *Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP 2000)*, August 2000. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[PFTV88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[SOHL$^+$00] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference (Volume 1)*. The MIT Press, 2000.

[VdV94] E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.