A Pattern Language for Parallel Application Programming

UF CISE Technical Report 98-019

Berna L. Massingill
CISE Department
University of Florida
blm@cise.ufl.edu

Timothy G. Mattson
Parallel Algorithms Laboratory
Intel Corporation
timothy.g.mattson@intel.com

Beverly A. Sanders
CISE Department
University of Florida
sanders@cise.ufl.edu

Abstract

Despite the best efforts of many researchers, the majority of programmers view parallel programming as too difficult to be practical. This is due in part to the fact that most research focuses on how concurrency in an algorithm is implemented rather than on the design of the parallel algorithm.

For software development in general, the use of design patterns has emerged as an effective way to help programmers design high-quality software. To be most useful, patterns that work together to solve design problems are collected into structured hierarchical catalogs called pattern languages. *A pattern language helps guide programmers through the whole process of application design and development.*

We believe that this approach can be usefully applied to parallel computing; that is, that we can make parallel programming attractive to general-purpose professional programmers by providing them with a pattern language for parallel application programming.

In this paper we provide an early look at our ongoing research into producing such a pattern language. We describe its overall structure, present a complete example of one of our patterns, and show how it can be used to develop a parallel global optimization application.

1. Introduction

Parallel hardware has been available for some decades now and is becoming increasingly mainstream. Software that takes advantage of these machines, however, is much rarer, largely because of the widespread and justified belief that such software is too difficult to write. Most research to date on making parallel machines easier to use has focused on the creation of parallel programming environments that hide the details of a particular computer system from the programmer, making it possible to write portable software; a good example of this sort of programming environment is MPI [11]. These environments have been successful overall in meeting the needs of the high-performance computing (HPC) community. Outside the HPC community, however, only a small fraction of programmers would even consider writing parallel software, so research into parallel programming environments cannot be considered a general success, and indeed this research seems to focus on the HPC community at the expense of the rest of the programming world.

The reasons for this state of affairs are complex. We believe the fault lies with the parallel

programming environments, but we do not believe that they can be fixed with yet another high-level abstraction of a parallel computer. To get professional programmers to write parallel software, we must give them a programming environment that helps them understand and express the concurrency in an algorithm. In other words, we need to solve this problem at the level of algorithm design.

At the same time, theoretical computer scientists have developed a substantial body of work that can be used to formally demonstrate correctness of parallel programs. Given the difficulty of debugging parallel programs, it would clearly be of benefit to programmers to help them avoid bugs in the first place, and what has been learned by the theoreticians could be of help in that regard — if it can be presented in a way that is accessible to professional programmers, which so far has not generally been the case.

Helping programmers design high-quality software is a general problem in software engineering. Currently, one of the most popular solutions is based on *design patterns* [5]. A design pattern is a carefully-written solution to a recurring software design problem. These patterns are usually organized into a hierarchical catalog of patterns. Software designers can use such a pattern catalog to guide themselves from problem specifications to complete designs. This collection of patterns is called a *pattern language*. An effective pattern language goes beyond the patterns themselves and provides guidance to the designer about how to use the patterns.

Design patterns have had a major impact on modern software design. We believe the same approach can be used to help programmers write parallel software. In other words, we believe that we can attract the general-purpose professional programmer to parallel computing if we give them a pattern language for parallel application programming. We also believe that such a pattern language can incorporate some of what has been learned by the theoretical community in a way that benefits our target audience.

In this paper, we provide an early view of our ongoing research [10] into producing a pattern language for parallel application programming. We begin by looking at patterns in the context of related work on skeletons, frameworks, and archetypes. We then give an overview of our pattern language — its overall structure and the notation we use for expressing patterns — and present two example patterns. Finally, we show how our pattern language can be used to write a parallel application, using our *Partitioning* pattern to create a parallel global optimization program.

2. Previous work

Considerable work has been done in identifying and exploiting patterns to facilitate software development, on levels ranging from overall program structure to detailed design. The common theme of all of this work is that of identifying a pattern that captures some aspect of effective program design and/or implementation and then reusing this design or implementation in many applications.

Program skeletons. Algorithmic skeletons, first described in [3], capture very high-level patterns; they are frequently envisaged as higher-order functions that provide overall structure for applications, with each application supplying lower-level code specific to the application. The emphasis is on design reuse, although work has been done on implementing program skeletons.

Program frameworks. Program frameworks [4] similarly address overall program organization, but they tend to be more detailed and domain-specific, and they provide a range of low-level functions and emphasize reuse of code as well as design.

Design patterns. Design patterns, in contrast to skeletons and frameworks, can address design

problems at many levels, though many existing patterns address fairly low-level design issues. They also emphasize reuse of design rather than reuse of code; indeed, they typically provide only a description of the pattern together with advice on its application. One of the most immediately useful features of the design patterns approach is simply that it gives program designers and developers a common vocabulary, since each pattern has a name. Early work on patterns (e.g., [5]) dealt mostly with object-oriented sequential programming, but more recent work ([15]) addresses parallel programming as well, though mostly at a fairly low level.

Programming archetypes. Programming archetypes [2, 8] combine elements of all of the above categories: They capture common computational and structural elements at a high level, but they also provide a basis for implementations that include both high-level frameworks and low-level code libraries. A parallel programming archetype combines a computational pattern with a parallelization strategy; this combined pattern can serve as a basis both for designing and reasoning about programs (as a design pattern does) and for code skeletons and libraries (as a framework does). Archetypes do not, however, directly address the question of how to choose an appropriate archetype for a particular problem.

3. Our pattern language

A design pattern is a *solution to a problem in a context*. We find design patterns by looking at high-quality solutions to related problems; the design pattern is written down in a systematic way and captures the common elements that distinguish good solutions from poor ones.

We can design complex systems in terms of patterns. At each decision point, the designer selects the appropriate pattern from a pattern catalog. Each pattern leads to other patterns, resulting in a final design in terms of a web of patterns.

A structured catalog of patterns that supports this design style is called a *pattern language*. A pattern language is more than just a catalog of patterns; it embodies a design methodology and provides domain-specific advice to the application designer.

The pattern language described in this document is intended for application programmers who want to design new programs for execution on parallel computers. The top-level patterns help the designer decide how to decompose a problem into components that can run concurrently. The lower-level patterns help the designer express this concurrency in terms of parallel algorithms and finally at the lowest level in terms of the parallel environment's primitives. These lower levels of our pattern language correspond to entities not usually considered to be patterns. We include them in our pattern language, however, so that the language is better able to provide a complete path from problem description to code, and we describe them using our pattern notation in order to provide a consistent interface for the programmer.

Our pattern language addresses only concurrency; it is meant to complement rather than replace other pattern-based design systems. Before programmers can work with this pattern language, they must understand the core objects, mathematics, and abstract algorithms involved with their problems. In this pattern language, we refer to design activities that take place in the problem domain and outside of this pattern language as *reasoning in the problem space*. The pattern language will use the results from problem space reasoning, but it will not help the programmer carry it out.

Before plunging into the structure of our pattern language, we offer one additional observation. In addition to facilitating reuse of code and design, patterns allow for a certain amount of "proof reuse": A pattern can include a careful specification for its application-specific parts, such that a use of the pattern that meets that specification is known to be correct by virtue of a proof

done once for the pattern. Patterns that include implementation can also have their implemented parts proved correct. Thus, patterns can serve as a vehicle for making some results from the theoretical community accessible and useful for programmers; our pattern language takes advantage of this.

3.1. Structure of our pattern language

Our pattern language is organized around four core design spaces arranged in a linear hierarchy, as shown in Figure 1. The higher-level spaces correspond to an abstract view of parallel programming, independent of particular target environments; lower-level spaces are more implementation-specific, with the lowest-level space a pattern-based description of the low-level building blocks used to construct parallel programs in particular target environments. To use our pattern language, application designers begin at the top level and work their way down, transforming problem descriptions into parallel algorithms and finally into parallel code. The remainder of this section describes our core design spaces. Observe that these design spaces, like all the components of our pattern language, are themselves patterns, each serving to organize the elements in the corresponding space.

The FindingConcurrency design space

This design space is concerned with structuring the problem to *expose exploitable concurrency*; it helps the programmer identify concurrently-executable units of work into which the problem can be divided. The designer working at this level focuses on high-level algorithmic issues and reasons about the problem to expose potential concurrency. Experienced parallel programmers may be able to skip this level and move directly to the *AlgorithmStructure* level. This space includes the following patterns.

DecompositionStrategy. This pattern helps a designer decide how to decompose a problem into concurrently-executable units of work; choices include task-based decomposition, data-based decomposition, or a combination of the two.

DependencyAnalysis. This pattern helps a designer analyze the dependencies between the tasks identified in the previous pattern.

CoordinationFramework This pattern helps a designer use the results of the previous two patterns to choose an overall structure for the parallel application from among those in the *AlgorithmStructure* design space.

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 at the *FindingConcurrency* level. Patterns in this space describe overall strategies for exploiting concurrency. This space includes the following patterns.

AsynchronousComposition. The problem is decomposed into a set of tasks that interact through asynchronous events.

PipelineProcessing. The problem is decomposed into an ordered set of tasks connected by data dependencies.

IterationSplitting. The parallelism is expressed in terms of splitting loop iterations between threads of execution. This is sometimes referred to as “loop splitting”, and it comes in two related forms: *replicated data*, in which data dependencies are pulled outside of the loop and shared data is replicated and recombined at the end of the loop; and *shared memory*, in which the data dependency is in the loop and explicitly managed.

If there are no dependencies between the loop iterations, this pattern is more properly thought of as an instance of the *Partitioning* pattern.

Partitioning. The problem is decomposed into a set of independent tasks. Most algorithms based on task queues and random sampling are instances of this pattern.

BalancedTree. The problem is mapped onto a balanced tree, and processing occurs to and from the root in k stages (where the problem size is on the order of n^k for an n-ary tree).

DivideAndConquer. 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.

GeometricDecomposition. The problem space is decomposed into discrete subspaces; the problem is then solved by computing solutions for the subspaces, with interaction largely taking place at subspace boundaries. Many instances of this pattern can be found in scientific computing, where it is useful in parallelizing grid-based computations, for example.

The SupportingStructures design space

This design space is concerned with lower-level algorithmic elements used to implement patterns in the *AlgorithmStructure* space. Two important groups of patterns in this space are those that represent program-structuring constructs (such as *ForkJoin*) and those that represent commonly-used shared data structures (such as *SharedQueue*). We observe again that many of the elements of this design space are not usually thought of as patterns, and indeed some of them (*SharedQueue*, for example) would ideally be provided to the programmer as part of a framework of reusable software components. We nevertheless document them as patterns, for two reasons: First, as noted earlier, documenting all the elements of our pattern language as patterns provides a consistent notation for the programmer. Second, the existence of such pattern descriptions of components provides guidance for programmers who might need to create their own implementations. The following are examples of patterns in this space.

SPMD (single program, multiple data). In this program-structuring pattern, the computation consists of n processes or threads executing in parallel. All n processes or threads execute the same program code, but each operates on its own set of data.

ForkJoin. In this program-structuring pattern, 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. Programs that make use of this pattern often compose multiple instances of it in sequence, as in the example in Section 5.

SharedQueue. 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.

The ImplementationMechanisms design space

This design space is concerned with how the patterns of the higher-level spaces are mapped into particular programming environments. We use it to provide pattern-based descriptions of common mechanisms for process/thread management (e.g., creating or destroying processes/threads) and process/thread interaction (e.g., monitors, semaphores, barriers, or message-passing). Patterns in this design space, like those in the *SupportingStructures* space, describe entities that strictly speaking are not patterns at all. As noted previously, however, we include them in our pattern language to provide a complete path from problem description to code, and we document them using our pattern notation for the sake of consistency. The following are examples of patterns in this space.

Forall. This pattern represents programming constructs that have the effect of creating multiple processes or threads, all executing the same code in parallel, with implicit synchronization such that the creating process or thread continues only after all the created processes or threads have terminated. An example of such a construct is the `PARALLEL DO` directive of OpenMP [14].

Spawn. This pattern represents programming constructs that have the effect of creating a process or thread that executes independently of its creator, with any synchronization between them needing to be provided explicitly.

Barrier. This pattern represents barrier synchronization, in which all processes or threads must arrive at the barrier before any can proceed beyond it. An example of such a construct is the `BARRIER` directive of OpenMP [14].

MessagePassing. This pattern represents message-passing (point-to-point or over channels), which incorporates aspects of both communication and synchronization.

3.2. Notation for our pattern language

We encode our patterns in a consistent format based on that of [5], with elements as described in this section. We will ultimately present the whole collection of patterns making up our language in the form of a collection of Web-accessible documents connected by hyperlinks.

Intent

This section contains a brief statement of the problem solved by this pattern. The goal of this section is to make it easy for an application designer scanning through a number of patterns to decide quickly which pattern fits the problem to be solved.

Also Known As

This section lists other names by which a pattern is commonly known.

Motivation

A pattern, recall, is defined as a "solution to a problem in a context". This section is where we describe the context in which one would use this pattern; it explains why a designer would use this pattern and what background information should be kept in mind when using it.

Applicability

When writing down a pattern, one of the key goals is to give the application designer the information needed to quickly decide which patterns to use. This is so important that we do it in two sections, the *Applicability* section and the *Restrictions* section. The first (this section) discusses, at a high level, when the pattern can be used. The goal of this section is to help the designer decide whether the pattern really fits the problem to be solved.

Restrictions

In this section we provide details to help designers ensure that they are using the pattern safely. Ideally, if the restrictions are followed faithfully, the designer should feel confident that the use of this pattern will work. Thus, this section states restrictions carefully and completely, and it fully discusses unusual boundary conditions.

Where appropriate, this section contains a hyperlink to a *Supporting Theory* section. This supporting section provides a more rigorous theoretical justification for the guidelines to safe use of the pattern.

Participants

This section describes the components whose interaction defines the pattern; i.e., it looks "inside the box" of the pattern. These components can be other patterns in the language or more loosely-defined entities.

Structure

This section describes how the participants interact to define this pattern. Currently we provide only a textual description, but eventually we will explore using a systematic form of graphical representation.

Collaborations

This section describes how the pattern works with other patterns to solve a larger problem; i.e., it looks "outside the box" of the pattern.

Consequences

Every design decision has consequences; there are advantages and disadvantages associated with the use of any pattern. The designer must understand these issues and make trade-offs between them. In this section, we give designers the information they need to make these trade-offs intelligently.

Implementation

This section explains how to implement the pattern, usually in terms of patterns from lower-level design spaces. The discussion focuses on high-level considerations common to all or most programming environments.

Sample Code

Programmers learn by example. In this section, we support this mode of learning by providing an implementation or implementations of the pattern in a particular programming environment. For patterns in higher-level design spaces, we often provide simply a pseudocode implementation; for patterns in lower-level design spaces, we provide sample code based on one or more popular programming environments such as OpenMP [14], MPI [11], or Java [1].

Known Uses

This section describes contexts in which the pattern has been used, where possible in the form of literature references.

Related Patterns

This section lists patterns related to this pattern. In some cases, a small change in the parameters of the problem can mean that a different pattern is indicated; this section notes such cases.

4. Example patterns

This section presents two example patterns: the *Partitioning* pattern from our *AlgorithmStructure* space in full, and an abbreviated version of the *SharedQueue* pattern from our *SupportingStructures* space.

4.1. The Partitioning pattern

This section contains the full text of the *Partitioning* pattern.

Intent

This pattern is used to describe concurrent execution of a collection of independent tasks. Parallel algorithms that use this pattern are sometimes called *embarrassingly parallel*, since once the tasks have been defined the potential concurrency is obvious.

Also Known As

- Master-Worker.
- Task Queue.

Motivation

Consider an algorithm that can be decomposed into many independent tasks. These *embarrassingly parallel* problems contain obvious concurrency that is trivial to exploit once these independent tasks have been defined. Nevertheless, while the source of the concurrency is obvious, taking advantage of it in a way that makes for efficient execution can be difficult.

The *Partitioning* pattern shows how to organize such a collection of tasks so they execute efficiently. The challenge is to organize the computation so that all processors finish their work at about the same time — that is, so that the computational load is balanced among processors.

This pattern automatically and dynamically balances the load. With this pattern, faster or less-loaded processors automatically do more work. When the amount of work required for each task cannot be predicted ahead of time, this pattern produces a statistically optimal solution.

Applicability

Use the *Partitioning* pattern when:

- The problem consists of independent tasks.
- The startup cost for initiating a task is much less than the cost of the task itself.
- The number of tasks is much greater than the number of processors to be used in the parallel computation.
- The effort required for each task or the processing performance of the processors varies unpredictably. This unpredictability makes it very difficult to produce an optimal static work distribution.

Restrictions

Summary of supporting theory. The *Partitioning* pattern is applicable when what we want to compute is a $solution(P)$ such that

$$\begin{aligned} solution(P) \;=\; & f(subsolution(P,0), \\ & subsolution(P,1), \ldots, \\ & subsolution(P,N-1)) \end{aligned}$$

such that for $i \neq j$, $subsolution(P,i)$ does not depend on $subsolution(P,j)$. That is, the original problem can be decomposed into a number of *independent* subproblems such that we can solve the whole problem by solving all of the subproblems and then combining the results. We could code a sequential solution thus:

```
Problem P;
Solution subsolutions[N];
Solution solution;
for (i=0; i<N; i++) {
  subsolutions[i] =
    compute_subsolution(P, i);
}
solution =
  compute_f(subsolutions);
```

If function `compute_subsolution()` modifies only local variables, it is straightforward to show (as for example in [7]) that the sequential composition implied by the `for` loop in the preceding program can be replaced by any combination of sequential and parallel composition without affecting the result. That is, we can partition the iterations of this loop among available processors or threads in whatever way we choose, so long as each is executed exactly once.

Observe that in this sequential solution each subsolution is saved in a distinct array element, allowing us to claim that computation of the subsolutions is completely independent. If instead we want to accumulate the subsolutions into a shared data structure such as a list or queue, the situation is slightly more complicated. Concurrency is still possible, however, if the order in which subsolutions are added to the shared data structure does not affect the result, as for example if the shared data structure represents an unordered set or list.[1]

Restrictions on when to apply. The key restriction on applying this pattern is that it must be possible to solve the subproblems into which we partition the original problem *independently*. Also, if the subsolution results are to be collected into a shared data structure, it must be the case that the order in which subsolutions are placed in this data structure does not affect the result of the computation.

Guaranteeing implementation correctness. Based on the preceding discussion, the keys to exploiting available concurrency while maintaining program correctness are as follows.

- *Solve subproblems independently.* Computing the solution to one subproblem must not interfere with computing the solution to another subproblem. This can be guaranteed if the code that solves each subproblem does not modify any variables shared between processes or threads.
- *Solve each subproblem exactly once.* How this is guaranteed depends on the implementation. For example, if a shared task queue is used to keep track of which subproblems have been solved, it must be implemented correctly, such that concurrent access does not result in, for example, the same task being dequeued twice. This can be ensured by implementing the task queue as an instance of the *SharedQueue* pattern (Section 4.2).
- *Correctly save subsolutions.* This is trivial if each subsolution is saved in a distinct variable, since there is then no possibility that the saving of one subsolution will affect subsolutions computed and saved by other tasks. If the subsolutions are to be collected into a shared data structure, then the implementation must guarantee that concurrent access does not damage the shared data structure. This can be ensured by implementing the shared data structure as an instance of a "thread-safe" pattern such as *SharedQueue* or *SharedCounter*.
- *Correctly combine subsolutions.* This can be guaranteed by ensuring that the code to combine subsolutions does not begin execution until all subsolutions have been computed.

[1] Formal justification for this claim is presented in a separate *Supporting Theory*, not included in this paper because of space constraints.

Participants

The participants in this pattern are the independent tasks.

Structure

The *Partitioning* pattern views a computation as a collection of independent tasks. The structure of a program using this pattern includes the following three parts:

- A definition of the tasks within the collection.
- A way to select the next task to carry out.
- A mechanism to detect completion of the tasks and to terminate the computation.

Collaborations

This pattern can be combined with others (or with itself) in sequence, as in the example in Section 5.

Consequences

The *Partitioning* pattern has some powerful benefits. First, parallel programs that utilize this pattern are among the simplest of all parallel programs. If the independent tasks correspond to individual loop iterations and these iterations do not share data dependencies, parallelization can be easily implemented with a parallel loop directive.

With some care on the part of the programmer, it is possible to implement programs with this pattern that automatically and dynamically adjust the load between processors. This makes the *Partitioning* pattern popular for programs designed to run on parallel computers built from networks of workstations.

This pattern is particularly valuable when the effort required for each task varies significantly and unpredictably. It also works particularly well on heterogeneous networks, since faster or less-loaded processors naturally take on more of the work.

The downside, of course, is that the whole pattern breaks down when the tasks need to interact during their computation. This limits the number of applications where this pattern can be used.

Implementation

There are many ways to implement this pattern. One of the most common is to collect the tasks into a queue (the *task queue*) shared among processes. This task queue can then be implemented using the *SharedQueue* pattern (Section 4.2).

Master-Worker versus SPMD. Frequently this pattern is implemented using two types of processes, *master* and *worker*. There is only one master process; it manages the computation by:

- Setting up or otherwise managing the workers.
- Creating and managing a collection of tasks (the task queue).
- Consuming results.

There can be many worker processes; each contains some type of loop that repeatedly:

- Removes the task at the head of the queue.
- Carries out the indicated computation.
- Returns the result to the master.

A common variation is to use an SPMD program with a global counter to implement the task queue pattern. This form of the pattern does not require an explicit master.

Termination. Termination can be implemented in a number of ways. One approach is for the master or a worker to detect the last task and then create a poison pill. The poison pill is a special task that tells all the other workers to terminate. Another approach is for the master to count results; when it detects that all results have been delivered, it halts the workers.

Correctness considerations. See the *Restrictions* section.

Efficiency considerations.

- If possible, put the longer tasks at the beginning of the queue. This ensures that there will be work to overlap with their computation.

Sample Code

Master-Worker example. Consider a problem consisting of N independent tasks. Assume we can map each task onto a sequence of simple integers ranging from 0 to $N - 1$. Further assume that the effort required by each task varies considerably and is unpredictable.

The code in Figure 2 and Figure 3 uses the *Partitioning* pattern to solve this problem. We implement the task queue as an instance of the *SharedQueue* pattern (see Section 4.2). To keep count of how many tasks have been completed

we need a shared counter that can be safely accessed by multiple processes or threads, which we could implement as an instance of the *SharedCounter* pattern (briefly described in Section 3.1). The master process, shown in Figure 2, initializes the task queue and the counter. It then forks the worker threads and waits until the counter indicates that all the workers have finished. At that point it consumes all the results and then kills the workers. The worker process, shown in Figure 3, is simply an infinite loop. Every time through the loop, it grabs the next task, does the indicated work storing the results into a global results array, and indicates completion of a result by incrementing the counter. The loop is terminated by the master process's killing the worker process. Note that we ensure safe access to key shared variables (the task queue and the counter) by implementing them using patterns from the *SupportingStructures* space. Note also that the overall organization of the master process is an instance of the *ForkJoin* pattern, briefly described in Section 3.1.

SPMD example. As an example of implementing this pattern without a master process, consider the following sample code using the TCGMSG message-passing library [6]. The library has a function called `NEXTVAL()` that implements a global counter. An SPMD program could use this construct to create a task-queue program as shown in Figure 4.

Known Uses

There are many application areas in which this pattern is useful. Many ray tracing codes use some form of partitioning with individual tasks corresponding to scan lines in the final image. Applications coded with the Linda coordination language are another rich source of examples of this pattern.

Parallel computational chemistry applications also make heavy use of this pattern. In the quantum chemistry code GAMESS, the loops over two electron integrals are parallelized with the TCGMSG task queue mechanism mentioned earlier. An early version of the Distance Geometry code, DGEOM, was parallelized with the Master-Worker form of the *Partitioning* pattern. These examples are discussed in [9].

Related Patterns

As mentioned earlier, this pattern can be thought of as a simple special case of the *IterationSplitting* pattern.

4.2. The SharedQueue pattern

This section presents an abbreviated version of the *SharedQueue* pattern.

Intent

This pattern represents an implementation of the familiar queue abstract data type (ADT) suitable for use by concurrently-executing processes or threads.

Motivation

Queues shared among processes are not uncommon in implementations of parallel algorithms. Careful thought, however, suggests that a naive sequential implementation of the queue ADT is not guaranteed to work if multiple concurrently-executing processes or threads have access to a queue. What is needed in this context is an implementation of the queue ADT that can be used by concurrent callers without problems — a "thread-safe" implementation, in other words.

Applicability

Use the *SharedQueue* pattern when concurrently-executing processes or threads must share a queue data structure (i.e., a data structure that implements the usual queue operations – enqueue, dequeue, etc.).

Restrictions

There are no particular constraints on applying this pattern; the difficulty is in implementing the queue ADT in such a way that it maintains the correct semantics even if its functions or methods are used by concurrently-executing processes or threads.

Collaborations

Outside agents interact with this pattern as with any other implementation of the queue ADT, through an interface that supports the following operations:

- *new*, which creates a new queue.
- *empty*, which indicates whether a queue is empty.
- *enqueue*, which adds a specified element to the tail of the queue.
- *dequeue*, which removes and returns the element at the head of the queue.

Implementation

As noted previously, this pattern would ideally already be implemented as part of a library of components compatible with the programmer's chosen programming environment. This section nonetheless indicates how to implement the pattern if no suitable implementation is available.

The key issue to be addressed in implementing this pattern is guaranteeing that the semantics of the queue are preserved even with multiple concurrent callers. (Careful thought will suggest how unrestricted concurrent access to the shared data structure that represents the queue could cause problems.) An easy, though probably not optimally efficient, way to do this is to ensure that at most one process at a time has access to the shared data structure that represents the queue.

(A complete version of this pattern, which we do not include because of space constraints, would also discuss more efficient implementations and would indicate which patterns in the *ImplementationMechanisms* space could be used to provide the requisite interprocess coordination.)

Sample Code

(A complete version of this pattern would provide one or more sample implementations.)

Known Uses

This pattern has many uses, for example in implementations of the *Partitioning* pattern.

Related Patterns

Related patterns include those representing other shared data structures, for example the *SharedCounter* pattern.

5. Example application: Global optimization

5.1. Problem description

As an example of our pattern language in action, we will look at a particular type of global optimization algorithm. This algorithm uses the properties of interval arithmetic [12] to construct a reliable global optimization algorithm.

Intervals provide an alternative representation of floating point numbers. Instead of a single floating point number, a real number is represented by a pair of numbers that bound the real number. The arithmetic operations produce interval results that are guaranteed to bound the mathematically "correct" result. This arithmetic system is robust and safe from numerical errors associated with the inevitable rounding that occurs with floating point arithmetic.

One can express most functions in terms of intervals to produce *interval extensions* of the functions. Values of the interval extension are guaranteed to bound the mathematically rigorous values

of the function. This fact can be used to define a class of global optimization algorithms [13] that find rigorous global optima. The details go well beyond the scope of this paper. The structure of the algorithm however, can be fully appreciated without understanding the details.

To make the presentation easier, consider the minimization of an objective function. This function contains a number of parameters that we want to investigate to find the values that yield a minimum value for the function. This problem is complicated by the fact that there may be 0 or many sets of such parameter values. A value that is a minimum over some neighborhood may in fact be larger than the values in a nearby neighborhood.

We can visualize the problem by associating an axis in a multidimensional plot with each variable parameter. A candidate set of parameters defines a box in this multidimensional space. We start with a single box covering the domain of the function. The box is tested to see if it can contain one or more minima. If the box cannot contain a minimum value, we reject the box. If it can contain a minimum value, we split the box into smaller subboxes and put them on a list of candidate boxes. We then continue for each box on the list until either there are no remaining boxes or the remaining boxes are sufficiently small. Pseudocode for this algorithm is given in Figure 5.

5.2. Parallelization using our pattern language

An experienced parallel programmer would immediately see this algorithm as an instance of our *Partitioning* pattern. For such a programmer, entering our language at the *Partitioning* pattern might be the right thing to do. Our pattern language, however, is targeted at professional programmers with little or no experience with parallel programming, and such programmers may need guidance from the pattern language to arrive at the *Partitioning* pattern.

Using the FindConcurrency design space

The first step for such a programmer is to find the concurrency in the algorithm, which is the domain of our *FindingConcurrency* design space. Entering our pattern language at that level, the programmer would use the *DecompositionStrategy* pattern. (Because of space constraints we do not give the text of the *FindConcurrency* patterns in this paper.) This pattern guides the programmer to the conclusion that a task-based decomposition is appropriate for this problem and that the natural unit of concurrency is the test of whether a box can contain a minimum. This is the most computationally intensive part of the problem, and the computation for any given box can be carried out independently of the other boxes.

Next, the programmer needs to understand the dependencies between concurrent tasks. Moving into the *DependencyAnalysis pattern*, he or she would see that interactions between tasks occur through the list. To prevent tasks from interfering with each other, the access to the list must be protected so that only one task at a time can read or write to the list.

A more subtle issue that would be exposed while working with the *DependencyAnalysis* pattern is the nature of the termination test. A test for termination of the algorithm cannot take place while any of the tasks are processing a box. This implies a partial order in the algorithm, which can be addressed by breaking it up into two phases: a *box processing phase* and a *termination test* phase.

With the fundamental decomposition in hand and the dependencies identified, the programmer can decide how to structure the concurrency so it can be exploited. Our pattern language guides this process via the *CoordinationFramework* pattern, which helps the programmer combine the decomposition strategy and the dependency analysis to choose an algorithm structure. Here, this pattern would guide the programmer to the *Partitioning* pattern.

Most of the literature concerned with the use of patterns in software engineering associates code, or constructs that will directly map onto code, with each pattern. It is important to appreciate, however, that patterns solve problems and that these problems are not always directly associated with code. In this case, for example, the first three patterns have not led to any code; rather they have helped the programmer reach an understanding of the best alternatives for structuring a solution to the problem. It is this guidance for the algorithm designer that is missing in most parallel programming environments.

Using the AlgorithmStructure design space

Having selected the *Partitioning* pattern, the programmer next uses it to help identify the key issues in designing the optimization algorithm. The computation defining a task is the test on whether a box can contain minima (i.e., the function `no_minima()`). To support the dual-phase structure and to ensure that subproblem results are correctly combined, two lists are needed: a *task list* and a *result list*.

As indicated earlier, the calculation proceeds in two phases. In the first phase, a set of tasks execute concurrently to test whether each box in the task list can contain any minima. If a box can, it is split into sub-boxes, and these sub-boxes are placed on the result list. In the second phase, the result list is checked to see if the termination condition has been satisfied. If so, the results are printed and the computation is finished. If not, the process is repeated with the result list becoming the task list for the next pass.

Reviewing the *Restrictions* section of the *Partitioning* pattern, we see that both lists need to be implemented as "thread-safe" shared data structures, since the task list will be used as a task queue and the result list will be used to collect subsolutions. Once again, an experienced parallel programmer would know this right away, but someone with less experience with parallel programming would need to be guided to this conclusion.

At this point the programmer has used our pattern language to decide the following:

- The program will be structured in two phases: box tests and termination detection.
- These two phases imply two box lists: an input list and a results list.
- The major source of productive concurrency is in the box tests.
- To ensure correctness, some data structures must be implemented using thread-safe shared-data-structure patterns.

He or she is now ready to start designing code.

Using the SupportStructures design space

As a first step in designing code, the programmer would probably consult the *SharedQueue* pattern to see how to use the shared queue library component. Ideally, the library would provide exactly what is needed, a shared-queue component that can be instantiated to provide a queue with elements of a user-provided abstract data type (in this case a box). If it did not, the pattern would indicate how the needed shared data structure could be implemented.

The programmer would next consider program structure, again guided by the *Partitioning* pattern. Given the two-phase structure in which the second phase (the termination test) is not computationally intensive, it makes sense to use a fork-join structure (the *ForkJoin* pattern, as described in the *Partitioning* pattern's *Implementation* section), in which a master process sets up the problem, initializes the queue, and then forks a number of processes or threads to test the boxes in the box list. Following the join, the master carries out the termination test sequentially and then as needed returns to process the list concurrently in the next cycle. The cycles continue until the termination conditions are met. Observe that the overall structure of the program is a composition

of a sequential program construct (a "while" loop) with an instance of the *ForkJoin* pattern.

Figure 6 shows pseudocode for the resulting design. Note that although this pseudocode omits major portions of the program (for example, details pertaining to the details of interval global optimization algorithms), it includes everything relevant to the parallel structure of the program.

Using the ImplementationMechanisms design space

Once the programmer has arrived at a design at the level of the pseudocode of Figure 6, he or she must then implement it in a particular programming environment, addressing whatever additional issues are relevant in that environment. (For example, implementing the *ForkJoin* pattern for a shared-memory multiprocessor differs substantially from implementing the same pattern for an cluster of workstations.) For this problem, all such issues are encapsulated in the *ForkJoin* and *SharedQueue* patterns, so the programmer can consult the *Implementation* section of these patterns for guidance on how to implement them for the desired environment (and recall, the *SharedQueue* pattern would ideally already be implemented and available as a library component). These patterns in turn guide the programmer into the *ImplementationMechanisms* design space, which provides lower-level and more environment-specific help, such that after review of the relevant patterns the programmer can finish the process of turning a problem description into finished code for the target environment.

6. Conclusions

In this paper, we described our ongoing research into the development of a pattern language for parallel application programming. Space constraints allowed us to provide only a limited view of our pattern language, consisting of a discussion of the design spaces used to organize the pattern language, the full text for one pattern, and abbreviated text for another pattern.

While a complete description of our pattern language could not be included in this paper, we did provide enough information to understand the structure of the patterns themselves and how we are organizing them into a pattern language. This structure is not the only way to organize patterns into a pattern language, but after much experimentation, it is the most effective organization we have found.

To demonstrate how the patterns would be used in a design problem, we discussed a global optimization application. It is true that this is an embarrassingly parallel application and thus may appear trivial to parallelize. The need for organizing the tasks into two phases, however, was not obvious, nor was the need for use of a shared queue data structure. While we have not conducted tests with programmers inexperienced in parallel computing, we believe that our pattern language effectively exposed these issues and would have helped such a programmer develop a correct design. Clearly, we have more work to do in order to test this hypothesis.

As mentioned earlier, this is an ongoing project. It is also an ambitious project that will undergo considerable evolution as we complete more patterns and apply them to more applications. Interested readers can follow our progress towards a pattern language for parallel application programming at ⟨ http://www.cise.ufl.edu/~blm/ParallelPatterns/ ⟩.

Acknowledgments

We gratefully acknowledge financial support from Intel Corporation, the NSF, and the AFOSR.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language: Second Edition*. Addison-Wesley, 1997.

[2] K. M. Chandy. Concurrent program archetypes. In *Proceedings of the Scalable Parallel Library Conference*, 1994.

[3] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[4] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*, pages 1–5. Addison-Wesley, 1995.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] R. J. Harrison. Portable tools and applications for parallel computers. *International Journal of Quantum Chemistry*, 40:847–863, 1991.

[7] B. L. Massingill. A structured approach to parallel programming. Technical Report CS-TR-98-04, California Institute of Technology, 1998. PhD thesis.

[8] B. L. Massingill and K. M. Chandy. Parallel program archetypes. Technical Report CS-TR-96-28, California Institute of Technology, 1996.

[9] T. G. Mattson. Scientific computation. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1996.

[10] T. J. Mattson, B. L. Massingill, and B. A. Sanders. A pattern language for parallel application programming, 1998. ⟨ http://www.cise.ufl.edu/~blm/ParallelPatterns/ ⟩.

[11] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[12] R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979.

[13] R. E. Moore, E. Hanson, and A. Leclerc. Rigorous methods for global optimization. In C. A. Foudas and P. M. Pardalos, editors, *Recent Advances in Global Optimization*, page 321, 1992.

[14] OpenMP Partners. The OpenMP standard for shared-memory parallel directives, 1998. ⟨ http://www.openmp.org ⟩.

[15] D. C. Schmidt. The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software. ⟨ http://www.cs.wustl.edu/~schmidt/ACE-papers.html ⟩, 1993.

Figures

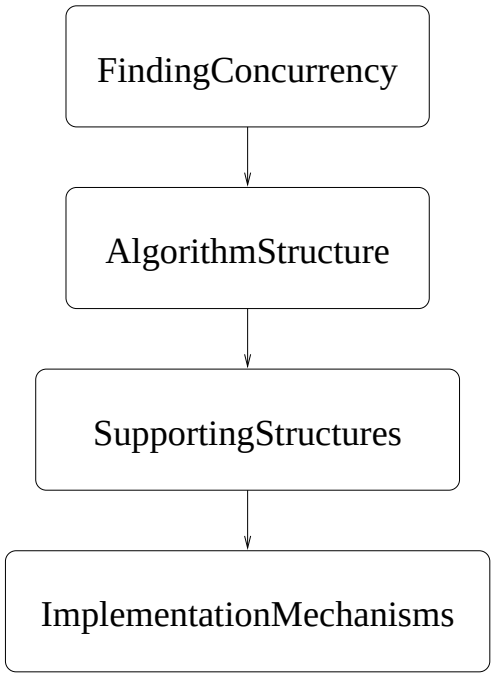

Figure 1. Structure of our pattern language.

```
#define Ntasks  500              /* Number of tasks       */
#define Nworkers  5              /* Number of workers     */
SharedQueue task_queue;          /* task queue            */
Results Global_results[Ntasks];  /* array to hold results */
SharedCounter done;              /* count finished results*/

void master()
{
   int Worker();

   // Create and initialize shared data structures
   task_queue = new SharedQueue();
   for (int i=0; i<N; i++)
      enqueue(&task_queue, i);
   done = new SharedCounter(0);

   // Create Nworkers threads executing function Worker()
   fork (Nworkers, Worker);

   // Wait for all tasks to be complete; combine results
   wait_on_counter_value (done, Ntasks);
   Consume_the_results (Ntasks);

   // Kill workers when done
   kill(Nworkers);
}
```

Figure 2. Sample code for Partitioning pattern master process.

```
int Worker()
{
  int i;
  Result res;

  While (TRUE) {
     i = dequeue(task_queue);
     res = do_lots_of_work(i);
     Global_results[i] = res;
     increment_counter(done);
  }
}
```

Figure 3. Sample code for Partitioning pattern worker process.

```
While (itask = NEXTVAL() < Number_of_tasks){
    DO_WORK(itask);
}
```

Figure 4. Sample code for Partitioning pattern SPMD process.

```
Interval_box B;
List_of_boxes L;
L = Initialize();

While (!done){
   B = get_next_box(L);
   if (no_minima (B))
      reject (B, L);
   else
      split_and_put (B, L);
   done = termination(L);
}

output(L);
```

Figure 5. Optimization algorithm (sequential version).

```
#define Nworkers N
SharedQueue<boxes> InList;
SharedQueue<boxes> ResList;
SharedCounter Workers_done
void main()
{
   int done = FALSE;
   InList = Initialize();

   While (!done) {
      Workers_done = new SharedCounter(0);

      // Create Workers to test boxes on InList and write
      // boxes that may have global minima to ResList
      Fork(Nworkers, workers);

      // Wait for the join (i.e. until all workers are done)
      Wait_on_counter_value(Workers_done, Nworkers);

      // Test for completion and copy ResList to InList
      done = termination(ResList, InList);
   }
   output(InList);
}

void function worker ()
{
   Interval_box B;

   While (!done) {
      B = dequeue(InList);

      // Use tests from Interval arithmetic to see
      // if the box can contain minima.  If so, split
      // into sub-boxes and put them on the result list.
      if (HasMinima (B))
         split_and_put (B, ResList);
      }

   increment_counter(Workers_done);
}
```

Figure 6. Optimization algorithm (parallel version).