Object–Oriented Pattern–Based Parallel Programming with Automatically Generated Frameworks

Steve MacDonald, Duane Szafron, and Jonathan Schaeffer
Department of Computing Science, University of Alberta
{stevem,duane,jonathan}@cs.ualberta.ca, http://www.cs.ualberta.ca/~{stevem,duane,jonathan}

Abstract

The CO_2P_3S parallel programming system uses design patterns and object–oriented programming to reduce the complexities of parallel programming. The system generates correct frameworks from pattern template specifications and provides a layered programming model to address both the problems of correctness and openness. This paper describes the highest level of abstraction in CO_2P_3S, using two example programs to demonstrate the programming model and the supported patterns. Further, we introduce *phased parallel design patterns*, a new class of patterns that allow temporal phase relationships in a parallel program to be specified, and provide two patterns in this class. Our results show that the frameworks can be used to quickly implement parallel programs, reusing sequential code where possible. The resulting parallel programs provide substantial performance gains over their sequential counterparts.

1 Introduction

Parallel programming offers potentially substantial performance benefits to computationally intensive applications. Using additional processors can increase the computational power that can be applied to large problems. Unfortunately, it is difficult to use this increased computational power effectively as parallel programming introduces new complexities to normal sequential programming. The programmer must create and coordinate concurrency. Synchronization may be necessary to ensure data is used consistently and to produce correct results. Programs may be nondeterministic, hampering debugging by making it difficult to reproduce error conditions.

We need development tools and programming techniques to reduce this added programming complexity. One such tool is a parallel programming system (PPS). A PPS can deal with complexity in several ways. It can provide a programming model that removes some of the added complexity. It can also provide a complete tool set for developing, debugging, and tuning programs. However, it is likely that there will still be some additional complexity the user must contend with when writing parallel programs, even with a PPS. We can ease this complexity by using new programming techniques, such as design patterns and object–oriented programming. Design patterns can help by documenting working design solutions that can be applied in a variety of contexts. Object–oriented programming has proven successful at reducing the software effort in sequential programming through the use of techniques such as encapsulation and code reuse. We want to apply these benefits to the more complex domain of parallel programming.

The CO_2P_3S[1] parallel programming system supports pattern–based parallel program development through framework generation and multiple layers of abstraction [7]. The system can be used to parallelize existing sequential code or write new, explicitly parallel programs. This system automates the use of a selected set of patterns through *pattern templates*, an intermediary form between a pattern and a framework. These pattern templates represent a pattern where some of the design alternatives are fixed and others are left as user–specified parameters. Once the template has been fully specified, CO_2P_3S generates a framework that implements the pattern in the context of the fixed and user–supplied design parameters. Within this framework, we introduce sequential *hook methods* that the user can implement to insert application–specific functionality. In the CO_2P_3S programming model, the higher levels of abstraction emphasize correctness and reduce the probability of programmer errors by providing the parallel structure and synchronization in the framework such that they cannot be modified by the user. The lower layers emphasize *openness* [13], gradually exposing low–level implementation details and introducing more opportunities for per-

[1] Correct Object–Oriented Pattern–based Parallel Programming System, pronounced "cops".

formance tuning. The user can work at an appropriate level of abstraction based on what is being tuned.

The key to reducing programmer errors is the decomposition of the generated framework into parallel and sequential portions. The framework implements both the parallel structure of the program and the synchronization for the execution of the hook methods. Neither of these attributes of the program can be modified at the highest level of abstraction. This decomposition allows the hook methods to be implemented as sequential methods, so users can concentrate on implementing applications rather than worrying about the details of the parallelism.

We introduce *phased parallel design patterns* in this paper. Phased patterns are unique in that they express a temporal relationship between different phases of a parallel program. Although all patterns have temporal aspects (such as specific sequences of method invocations), the intent of phased patterns is to deal with changing concurrency requirements for different phases of a parallel program. We introduce two such patterns, the Method Sequence and the Distributor, both of which are new. The Method Sequence can be used to implement phased algorithms, explicitly differentiating between the different phases of a parallel algorithm. This pattern recognizes that efficiently parallelizing a large program will likely require the application of several parallel design patterns. The Distributor pattern allows the user to selectively parallelize a subset of methods on an object, acknowledging that not all operations may have sufficient granularity for parallel execution.

We also introduce a structural pattern, the Two–Dimensional Mesh. The Mesh was written in an object–oriented fashion to fit within the CO_2P_3S system.

To demonstrate the use of the CO_2P_3S system, we present the development of two example programs. Our first example, reaction–diffusion texture generation [16], uses the Mesh pattern to simulate the reaction and diffusion of two chemicals over a two–dimensional surface. The second program implements the parallel sorting by regular sampling algorithm [12] using the Method Sequence and Distributor patterns. Both programs are implemented using the facilities provided at the highest level of abstraction in CO_2P_3S to demonstrate the utility of our patterns and the utility of the frameworks we generate to support our pattern templates.

The research contributions of this paper are:

- A layered parallel programming model that presents several different programming abstractions to the user. Each layer emphasizes different concerns, starting with program correctness at the highest level of abstraction, and providing openness and performance tuning at lower levels.
- The generation of a correct parallel framework from a pattern template specification, coupled with correctness guarantees by preventing the user from modifying the structure of the framework at the highest level of abstraction.
- An easy–to–use programming model that allows users to implement a parallel program by writing a small amount of sequential code to reuse their existing application code.
- An object–oriented pattern–based tool for parallel programming. We also introduce a new type of parallel design pattern, called phased patterns, to express time–related aspects of a parallel program.
- A demonstration of the benefits of the CO_2P_3S system using two example programs. These examples illustrate the use of the highest level of abstraction in the CO_2P_3S model and demonstrate the benefits of using a high–level tool for parallel programming.

2 Overview of the CO_2P_3S System

This section presents a brief overview of the CO_2P_3S system. A more detailed description can be found in [7].

The CO_2P_3S parallel programming system provides three levels of abstraction that can be used in the development of parallel programs. These abstractions provide a programming model that allows the programs to be tuned incrementally, allowing the user to develop parallel programs where performance is more directly commensurate with effort. These abstractions, from highest to lowest, are:

Patterns Layer The user selects a parallel design pattern template from a palette of supported templates. These templates represent a partially specified design pattern, where some of the design tradeoffs have been fixed. Each pattern template has several parameters that must be supplied before the template can be instantiated, allowing the user to specialize the framework implementing the pattern for its intended use. Instantiating the pattern template generates code that forms a framework for the template. The code consists of one or more

abstract classes that implement the parallel structure of the pattern template together with concrete subclasses, as well as any required collaborator classes. The framework code is customized in two ways: application–specific pattern template parameters and user–supplied implementations of specific sequential hook methods in the concrete subclasses (using the Template Method design pattern [3]). Since the user cannot modify the parallel structure at this layer, parallel correctness is ensured. A complete program consists of either a single framework or several frameworks composed together.

Intermediate Code Layer This layer provides a high–level, object–oriented, explicitly–parallel programming language, a superset of an existing OO language. The user manipulates both parallel structural code and application–specific code using this intermediate language.

Native Code layer The intermediate language is transformed into code for a native object–oriented language (such as Java or C++). This code provides all libraries used to implement the intermediate code from the previous layer. The user is free to use the provided libraries and language facilities to modify the program in any way.

Users can move down through the different abstractions, selecting a suitable layer based on the desired performance of their applications or on how comfortable they are with a given abstraction.

Several critical aspects of the framework are demonstrated by the use of hook methods for introducing application–specific functionality. First, the parallel structural code cannot be modified in the Patterns Layer, which allows us to make correctness guarantees about the parallel structure of the program. Second, it allows users to concentrate on implementing their applications without worrying about the structure of the parallelism. Also, by ensuring that the structural code provides proper synchronization around hook method invocations, the user can write sequential code without having to take into account the parallelism provided by the framework. Lastly, we provide suitable default implementations of the hook methods in the abstract classes of the framework. These default methods permit the framework to be compiled and executed immediately after it is generated, without implementing any of the hook methods. The program will execute with a simple default behaviour. This provides users with a correct implementation of the structure of the pattern before they begin adding the hook methods and tuning the program.

The CO_2P_3S system currently supports several parallel design patterns through its templates, which also use a group of sequential design patterns. These patterns are written specifically for solving design problems in the parallel programming domain. The patterns used in this paper are:

Method Sequence This new pattern supports the creation of phased algorithms by invoking an ordered sequence of methods on a Facade [3] object.

Distributor This new pattern supports data–parallel style computations by forwarding a method from a parent object to a fixed number of child objects, all executing in parallel.

Two–Dimensional Mesh This pattern supports iterative computations for a rectangular two–dimensional mesh, where a surface is decomposed into a set of regular, rectangular partitions.

A more detailed description of these patterns, along with the other patterns supported by CO_2P_3S, can be found in our pattern catalogue [6].

The context of our patterns is the architecture and programming model of CO_2P_3S. Therefore, we have made several changes to the structure of the basic design pattern documentation [3]. Since our patterns are for the parallel programming domain, the pattern description includes concurrency–related specifications, such as synchronization and the creation of concurrent activity. From the pattern, we produce a CO_2P_3S pattern template and an example framework, which we also describe in the pattern document. While these two sections are not strictly pattern specifications, they illustrate the use of the pattern while documenting the CO_2P_3S templates and frameworks. So while we have added CO_2P_3S–specific sections to our pattern descriptions, we have preserved the instructional nature of design patterns.

We note that our patterns still represent abstract design solutions. From the abstract pattern, we fix some of the design tradeoffs and allow the user to specify the remaining tradeoffs using parameters. This intermediate form, which is less abstract than a pattern but less concrete than a framework, is called a pattern template. We use a fully specified pattern template to generate parametrically related families of frameworks, with each framework in the family implementing the same basic pattern structure but specialized with the user–supplied parameters. However, the pattern templates are a design construct used to specify a framework; the templates do not contain nor provide code themselves. The generated

frameworks provide reusable and extendible code implementing a specific version of the pattern. These frameworks take into account our decomposition of a program into its sequential and parallel aspects. The ability of the user to modify a framework is dictated by the level of abstraction that the user is using.

In creating the pattern templates, we try to fix as few of the design tradeoffs as possible. However, in fixing any part of the design, we will create situations in which it will be necessary to modify certain elements of a program to fit within the limitations we impose. For the example programs that follow, we discuss the design in terms of the patterns, and discuss the implementation in terms of the pattern templates and the frameworks. The implementation of the programs may need to be modified from the initial design to account for the fixed design tradeoffs in the pattern templates and frameworks.

3 Example Applications

In this section, we detail the design and implementation of two example programs. These examples demonstrate the applicability of our patterns to application design, and show how the pattern templates and generated frameworks can be used to implement the programs. The first example uses a reaction–diffusion texture generation program to show how we have reworked the Mesh pattern in an object–oriented fashion. The second example, parallel sorting by regular sampling, uses the Method Sequence and Distributor patterns. We also use this example to highlight the temporal nature of these two patterns and to show the composition of CO_2P_3S frameworks. The implementation of these two examples demonstrates that the Patterns Layer of CO_2P_3S can be used to write parallel programs with good performance. It also shows the benefits of a high–level object–oriented tool for parallel programming.

3.1 Reaction–Diffusion Texture Generation

Reaction–diffusion texture generation [16] can be described as two interacting LaPlace equations that simulate the reaction and diffusion of two chemicals (called *morphogens*) over a two–dimensional surface. This simulation, started with random concentrations of each morphogen on the surface, can produce texture maps that approximate zebra stripes. The problem uses Gauss–Jacobian iterations (each iteration uses the results from the previous iteration to compute the new values, without any successive overrelaxation) and is solved using straightforward convolution. The simulation typically executes until the total change in morphogen concentration over the surface falls below some threshold. We use a fully toroidal mesh, allowing the morphogens to wrap around the edges of the surface. The toroidal boundary conditions ensure that the resulting texture can be tiled on a display without any noticeable edges between tiles.

3.1.1 Parallel Implementation

An obvious approach to this problem is to decompose the two–dimensional surface into regular rectangular regions and to work on these regions in parallel. Our solution must be more complex because we cannot evaluate each region in isolation. Each point on the surface needs the concentration values from its neighbours to calculate its new value, so points on the edge of a region need data from adjoining regions. Thus, we require an exchange of region boundaries between iterations. Further, Gauss–Jacobian iterations introduces dependencies between iterations that need to be addressed by our parallel implementation of this problem.

3.1.2 Implementation in CO_2P_3S

Design and Pattern Specification The first step in implementing a CO_2P_3S program is to analyze the problem and select the appropriate design pattern. This process still represents the bottleneck in the design of any program. Given the requirements of our problem, the Mesh pattern is a good choice for several reasons. The problem is an iterative algorithm executed over a two–dimensional surface. Further, our approach is to decompose the surface into regular rectangular regions, a decomposition that is automatically handled by the framework for the Mesh pattern template.

The Mesh pattern consists of two types of objects, a collector object and a group of mesh objects. The structure of the Mesh is given in Figure 1. The collector object is responsible for creating the mesh objects, distributing the input state over the mesh objects, controlling the execution of the mesh objects, and collecting the results of the computation. The mesh objects implement an iterative mesh computation, a loop that exchanges boundaries with neighbouring mesh elements and computes the new values for its local mesh elements. The iterations continue until all the mesh objects have finished

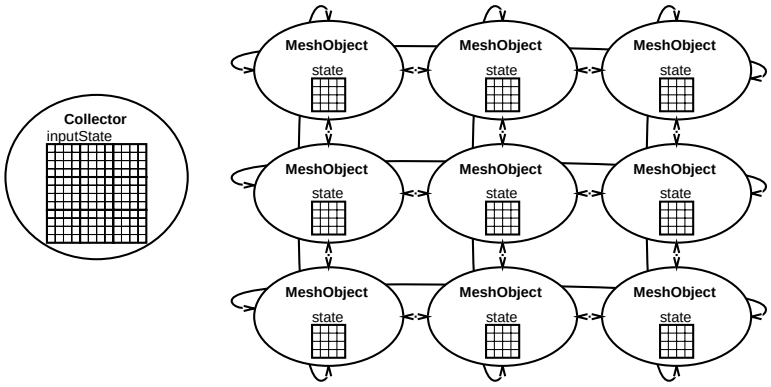


Figure 1: An object diagram for the structure of the Mesh pattern. The arcs indicate object references. The `Collector` object has references to all of the instances of `MeshObject`, but the arcs are omitted for clarity.

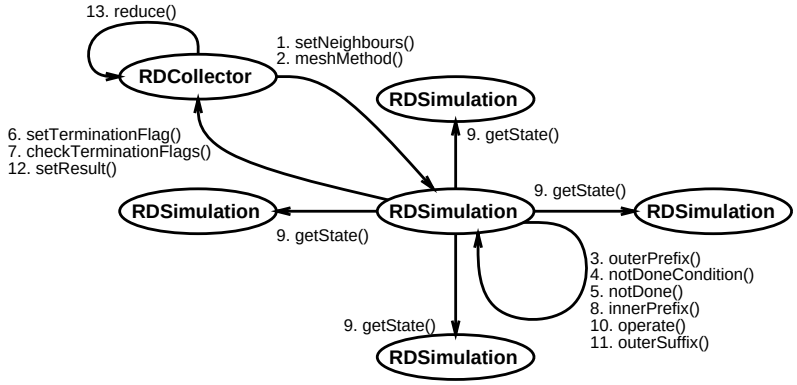


Figure 2: The method invocations in the Mesh implementation of the reaction–diffusion problem. The methods are executed in the order in which they are numbered. Calls 4 through 10 are the main loop of the mesh computation and are repeated until the computation has completed.

computing, when all of the morphogen concentrations in all of the mesh objects have stabilized in our example.

This pattern requires synchronization because of dependencies between iterations. Specifically, we need to ensure that the boundary elements for each mesh element have all been computed before exchanging them, to prevent a neighbour from using incorrect data in its next iteration. We also require synchronization for determining the termination condition, since all of the mesh objects must cooperate to decide if they have finished. This synchronization is necessarily pessimistic to handle the general case. Individual programs using this pattern may remove any unnecessary synchronization.

The Mesh pattern is not specific to the reaction–diffusion example. It can be used to implement other finite element calculations and image processing applications.

Once the pattern has been selected, the user selects the corresponding pattern template and fills in its parameters. For all pattern templates, the names of both the abstract and concrete classes for each class in the resulting framework are required. In CO_2P_3S, the user only specifies the concrete class names; the corresponding abstract class names are prepended with "Abstract". For our Mesh example, we specify `RDCollector` for the collector class and `RDSimulation` for the mesh class, which also creates the abstract classes `AbstractRDCollector` and `AbstractRDSimulation`. We further specify the type of the two–dimensional array that will be distributed over the mesh objects, which is `MorphogenPair` for the reaction–diffusion example. Finally, we specify the boundary conditions of the mesh, which is set to fully toroidal (where each mesh object has all four neighbours by wrapping around the edges of the mesh, as shown in Figure 1). We can select other boundary conditions (horizontal–toroidal, vertical–toroidal, and non–toroidal); we will see the ef-

fects of different conditions later in this section. The dimensions of the mesh are specified via constructor arguments to the framework. The input data is automatically block–distributed over the mesh objects, based on the dimensions of the input data and the dimensions of the mesh.

Using the Framework From the pattern template specification, the CO_2P_3S system generates a framework implementing the specific instance of the Mesh pattern given the pattern template and its parameters. The framework consists of the four classes given above with some additional collaborator classes. The sequence of method calls for the framework is given in Figure 2.

Once the framework is generated, the user can implement hook methods to insert application–specific functionality at key points in the structure of the framework. The selection of hook methods is important since we enforce program correctness at the Patterns Layer by not allowing the user to modify the structural code of the framework. The user implements the hook methods in the concrete class by overriding the default method provided in the abstract superclass in the framework. If the hook method is not needed in the application, the default implementation can be inherited.

To demonstrate the use of the hook methods, we show the main execution loop of the Mesh framework, as generated by CO_2P_3S, in Figure 3. The hook methods are indicated in bold italics. There are hook methods for both the collector and mesh objects in the Mesh framework. For the collector, the only hook method is:

`reduce()` This method, invoked after the mesh computation has finished, allows the user to perform a reduction on the results. By default, this method returns the input array updated with the results of the mesh computation.

The hook methods for the mesh objects are:

`outerPrefix()` This method is invoked before the mesh computation is started. It can be used for initializing the mesh object. By default, this method does nothing.

`notDone()` This method is used to determine if the mesh object has finished its local computation. Note that the computation finishes only when all mesh objects have finished. By default, this method returns `false`, indicating that the mesh object has finished its computation. This method is not directly invoked from the `meshMethod()` method. It is invoked indirectly from the `notDoneCondition()` method, which uses the result of this hook method in the `setTerminationFlag()` method to set the termination status of the current mesh object and then uses `checkTerminationFlags()` to calculate the global termination condition.

`innerPrefix()` This method is invoked first in the mesh computation loop, before the boundary exchange. It can be used for any precomputations required in the loop. By default, this method does nothing.

The operation methods These methods implement the mesh computation. There are nine methods that may be used depending on the boundary conditions. These are described below. By default, these methods do nothing. They are invoked indirectly from the `operate()` method. These methods replace the `innerSuffix()` method.

`outerSuffix()` This method is invoked after the mesh computation has finished but before results are passed back to the collector object. It can be used for any post–processing or cleanup required by the mesh object. By default, this method does nothing.

The implementation of the `operate()` method called in the code from Figure 3 invokes a subset of the nine operation methods given in Figure 4. The boundary conditions and the position of the mesh object determine which of the operation methods are used. For instance, consider the two meshes in Figure 5. For the fully toroidal mesh in Figure 5(a), there are no boundary conditions. Thus, only the `interiorNode()` hook method is invoked. For the horizontal–toroidal mesh in Figure 5(b), there are three different cases, one for each row. The mesh objects in the different rows, from top to bottom, invoke the `topEdge()`, `interiorNode()`, and `bottomEdge()` hook methods for the mesh operation. The implementation of the `operate()` method uses a Strategy pattern [3], where the strategy corresponds to the selected boundary conditions. This strategy is a collaborator class generated with the rest of the framework. It is also responsible for setting the neighbours of the mesh elements after they are created, using the `setNeighbours()` method (from Figure 2). At the Patterns Layer, the user does not modify this class. Each mesh object automatically executes the correct methods, depending on its location in the mesh.

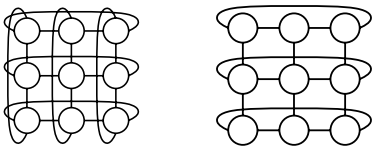
Now we implement the reaction–diffusion texture generation program using the generated Mesh framework.

```
public void meshMethod()
{
    this.outerPrefix() ;
    while(this.notDoneCondition()) {
        this.innerPrefix() ;
        MorphogenPair[][] leftState = left.getState() ;
        MorphogenPair[][] rightState = right.getState() ;
        MorphogenPair[][] upState = up.getState() ;
        MorphogenPair[][] downState = down.getState() ;
        this.operate(leftState, rightState, upState, downState) ;
    } /* while */
    this.outerSuffix() ;
    this.getCollector().setResult(this.getState()) ;
} /* meshMethod */
```

Figure 3: The main execution loop of a mesh. Hook methods are shown in bold italics.

```
void topLeftCorner(MorphogenPair[][] right, MorphogenPair[][] down) ;
void topEdge(MorphogenPair[][] right, MorphogenPair[][] left,
             MorphogenPair[][] down) ;
void topRightCorner(MorphogenPair[][] left, MorphogenPair[][] down) ;
void leftEdge(MorphogenPair[][] right, MorphogenPair[][] up,
             MorphogenPair[][] down) ;
void interiorNode(MorphogenPair[][] left, MorphogenPair[][] right,
             MorphogenPair[][] up, MorphogenPair[][] down) ;
void rightEdge(MorphogenPair[][] left, MorphogenPair[][] up,
             MorphogenPair[][] down) ;
void bottomLeftCorner(MorphogenPair[][] right, MorphogenPair[][] up) ;
void bottomEdge(MorphogenPair[][] left, MorphogenPair[][] right,
             MorphogenPair[][] up) ;
void bottomRightCorner(MorphogenPair[][] left, MorphogenPair[][] up) ;
```

Figure 4: The hook methods for the mesh operations.



(a) A fully toroidal mesh.

(b) A horizontal–toroidal mesh.

Figure 5: Example mesh structures.

First, we note that we do not need a reduction method, as the result of the computation is the surface computed by each region. Also, we do not require the `outerPrefix()` or the `outerSuffix()` methods. The `innerPrefix()` method is required because we have chosen to keep two copies of each morphogen array, a read copy for getting previous data and a write copy for update during an iteration. This scheme uses additional memory, but obviates the need to copy the array in each iteration. Each iteration must alternate between using the read and write copies, which is accomplished by reversing the references to the arrays in the `innerPrefix()` method. Given our fully toroidal mesh, we only need to implement the mesh operation in the `interiorNode()` hook method.

The `notDone()` method checks the local mesh state for convergence. Each mesh object returns a Boolean flag indicating if it has finished its local computation, and these flags are used to determine if all of the mesh objects have finished. The pattern template fixes the flags as Booleans, which does not allow the global termination conditions given in Section 3.1 to be implemented. Instead, our simulation ends when the change in morphogen concentration in each cell falls below a threshold. Although this restriction forced us to modify this program, it simplifies the pattern template specification and reduces the number of hook methods. This termination behaviour can be modified at the Intermediate Code Layer if a global condition must be implemented.

After completing the specification and implementation of the Mesh framework, the user must implement the code to instantiate the objects and use the framework. The Java code is given in Figure 6, where we use constants for the width and height of the data and the mesh, but these values could be obtained dynamically at run–time from a file or from the user.

3.1.3 Evaluation

In this section, we evaluate the Patterns Layer of CO_2P_3S using the reaction–diffusion texture generator. Our basis for evaluation is the amount of code written by the user to implement the parallel program and the run–time performance. These results are based on a Java implementation of the problem.

In the following discussion, we do not include any comments in counts of lines of code. All method invocations and assignments are considered one line, and we count all method signatures (although the method signatures for the hook methods are generated for the user).

The sequential version of the reaction–diffusion program was 568 lines of Java code. The complete parallel version, including the generated framework and collaborating classes, came to 1143 lines. Of that 1143 lines, the user wrote (coincidentally) 568 lines, just under half. However, 516 lines of this code was taken directly from the sequential version. This reused code consisted of the classes implementing the morphogens. This morphogen code had to be modified to use data obtained from the boundary exchange, whereas the sequential version only used local data. This modification required one method to be removed from the sequential version and several methods added, adding a total of 52 lines of code to the application. The only code that could not be reused from the sequential version was the mainline program. In addition, the user was required to implement the hook methods in the Mesh framework. These methods were delegated to the morphogen classes and required only a single line of code each.

We note that this case is almost optimal; the structure of the simulation was parallelized without modifying the computation. Also, the structure of the parallel program is close to the sequential algorithm, which is not always the case. These characteristics allowed almost all of the sequential code to be reused in the parallel version.

This program was executed using a native–threaded Java implementation from SGI (Java Development Environment 3.1.1, using Java 1.1.6). The programs were compiled with optimizations on and executed on an SGI Origin 2000 with 42 195MHz R10000 processors and 10GB of RAM. The Java virtual machine was started with 512MB of heap space. Results were collected for programs executed with the just–in–time (JIT) compiler enabled and then again with the JIT compiler disabled. Disabling the JIT compiler effectively allows us to observe how the frameworks behave on problems with increased granularity. Speedups are based on wall–clock time and are compared against a sequential implementation of the same problem executed using a green threads virtual machine. Note that the timings are only for the mesh computation; neither initialization of the surface nor output is included. The results are given in Table 1.

With the JIT enabled, the speedups for the program tail off quickly. As we add more processors, the granularity of the mesh computation loop falls and the synchronization hampers performance. The non–JIT version shows good speedups, scaling to 16 processors, showing the effects of increased granularity.

From this example, we can see that our framework promotes the reuse of sequential code; almost all of the morphogen code from the sequential program was reused in the parallel version. This reuse allowed the parallel version of the problem to be implemented with only a few new lines of code (52 lines). The performance of the resulting parallel application is acceptable with the JIT enabled, although the granularity quickly falls.

3.2 Parallel Sorting by Regular Sampling

Parallel sorting by regular sampling (PSRS) is a parallel sorting algorithm that provides a good speedup over a broad range of parallel architectures [12]. This algorithm is explicitly parallel and has no direct sequential counterpart. Its strength lies in its load balancing strategy, which samples the data to generate pivot elements that evenly distribute the data to processors.

The algorithm consists of four phases, illustrated in Figure 7. Each phase must finish before the next phase starts. The phases, executed on p processors, are:

1. In parallel, divide the input array into p contiguous lists and sort each list. Select $p - 1$ evenly spaced sample elements from each sorted list.
2. Select a designated processor to sort the entire set of sample elements. Then, choose $p - 1$ evenly spaced pivot values from the sample set.

```
public static void main(String[] argv)
{
    MorphogenPair[][] data = Main.initializeData(Main.dataWidth,
            Main.dataHeight) ;
    RDCollector collector = new RDCollector(Main.meshWidth,
            Main.meshHeight, data, Main.dataWidth, Main.dataHeight) ;
    /* Start the execution of the simulation.  */
    collector.Execute() ;
    /* Wait for and get the results.  */
    data = (MorphogenPair[][]) collector.getResults() ;
} /* main */
```

Figure 6: The code that starts the reaction–diffusion simulation using the Mesh framework.

Problem Size	JIT disabled				JIT enabled			
	2 proc.	4 proc.	8 proc.	16 proc.	2 proc.	4 proc.	8 proc.	16 proc.
1024×1024	1.96 11150 sec	3.72 5886 sec	6.93 3162 sec	12.39 1767 sec	1.61 2519 sec	2.88 1413 sec	4.49 906 sec	4.58 887 sec

Table 1: Speedups for the reaction–diffusion example. Wall clock times, rounded to the second, are also provided.

3. In parallel, partition each sorted list into p sublists using the pivot values.

4. In parallel, merge the partitions and store the results back into the array.

3.2.1 Parallel Implementation

The parallelism in this problem is clearly specified in the algorithm from the previous section. We require a series of phases to be executed, where some of those phases use a set of processors computing in parallel. For the parallel phases, a fixed number of processors execute similar code on different portions of the input data.

3.2.2 Phased Parallel Design Patterns

An interesting aspect of the PSRS algorithm is that the parallelism changes in different phases. The first and third phases are similar. The second phase is sequential. Finally, the last phase consists of two subphases (identified below), where each subphase has its own concurrency requirements. We need to ensure that this temporal relationship between the patterns can be expressed. In contrast, other parallel programming systems require the user to build a graph that is the union of all the possible execution paths that are used and leave it to the user to ensure they are used correctly. Alternately, the user must use a pipeline solution, where each pipe stage implements a phase (as in Enterprise [9]). However, the real strength of a pipeline lies in algorithms where multiple requests can be concurrently executing different operations in different pipe stages. Further, a pipeline suggests that a stream of data (or objects in an OO pipeline) is being transformed by a sequence of operations. A phased algorithm may transform its inputs or generate other results, depending on the algorithm.

A further temporal aspect of this algorithm is when to use parallelism. Sometimes we would like to use the same group of objects for both parallel and sequential methods. For instance, some methods may not have enough granularity for parallel execution. Sometimes, as in the second phase of PSRS, we may need to execute a sequential operation on data contained in a parallel framework. This kind of control can be accommodated by adding sequential methods in the generated framework. These methods would use the object structure without the concurrency. In implementing these methods, the user must ensure that they will not interfere with the execution of any concurrent method invocations.

3.2.3 Implementation in $\mathbf{CO_2P_3S}$

Design and Pattern Specification

Method Sequence In the PSRS algorithm, the phases stand out as the first concern. The algorithm suggests four phases. We note, though, that the last phase contains a data dependency. We must ensure that the merging is complete before we can begin writing results back to the original array. Otherwise, the merging phase can

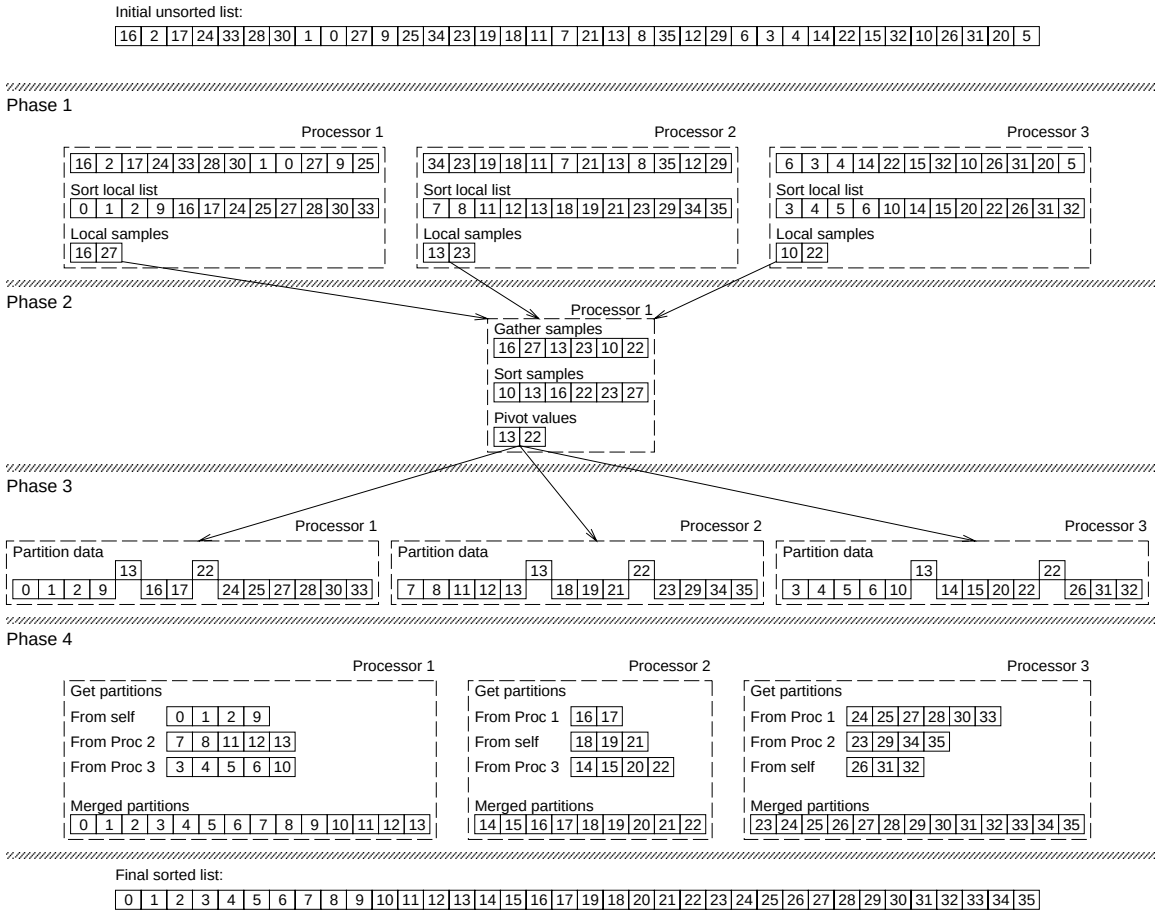


Figure 7: An example of parallel sorting by regular sampling.

read incorrect data. From this observation, we rewrite the fourth phase of PSRS as two subphases:

4.1 Merge the partitions in a temporary buffer.

4.2 Store the results in the original array by copying the temporary buffer.

To implement a series of phases in a program, we can use the Method Sequence pattern. In our example, we have identified two different sets of phases so we apply this pattern twice. The first application of the Method Sequence pattern implements the four phases from the previous section. We apply the pattern again in the implementation of the last phase, to execute the phases identified above. Alternately, we could rewrite the original algorithm as five phases and apply the pattern once. However, our solution is consistent with the original algorithm and also helps us demonstrate the composability of our frameworks in the next subsection.

The Method Sequence pattern is a specialization of the Facade pattern [3] that adds ordering to the methods of the Facade. It consists of two objects, a sequence object and an instance of the Facade pattern [3]. The sequence object holds an ordered list of method names to be invoked on the Facade object. These methods have no arguments or return values. The Facade object supplies a single entry point to the objects that collaborate to execute the different phases. The Facade typically delegates its methods to the correct collaborating object, where these methods implement the different phases for the sequence object. Each phase is executed only after the previous phase has finished. The Facade object is also responsible for keeping any partial results generated by one phase and used in another (such as the pivots generated in the second phase of PSRS and used in the third phase). We include the Facade object for the general case of the Method Sequence pattern, where there may be different collaborating objects implementing different phases of the computation. Without the Facade object, the sequence object would need to manage both the method list and the objects to which the methods are delegated, making the object more complicated.

The Method Sequence pattern has other uses beyond this example. For example, it is applicable to programs that can be written as a series of phases, such as LU factorization (a reduction phase and a back substitution phase).

After designing this part of the program, the user selects the Method Sequence pattern template and fills in the parameters to instantiate the template. For this pattern template, the user specifies the names of the concrete classes for the sequence and Facade classes, and an ordered list of methods to be invoked on the Facade object. Again, the abstract superclasses for both classes have "Abstract" prepended to the concrete class names.

Distributor Now we address the parallelism in the first, third, and last phases. Each of these phases require a fixed amount of concurrency (p processors). If we attempt to vary the number of processors for different phases, we will generate different data distributions that will cause problems for the operations. Further, the processors operate on the same region of data in the first and third phases. If we can distribute the data once to a fixed set of processors, we can avoid redistribution costs and preserve locality. The last phase requires a redistribution of data, but again it must use the same number of processors as used in the previous parallel phases. Similarly, the two subphases for the last phase share a common region, the temporary buffer. It is also necessary for the concurrency to be finished at the end of each phase because of the dependencies between the phases.

Given these requirements, we apply the Distributor pattern. This pattern provides a parent object that internally uses a fixed number of child objects over which data may be distributed. In the PSRS example, the number of children corresponds to the number of processors p. All method invocations are targeted on the parent object, which controls the parallelism. In this pattern, the user specifies a set of methods that can be invoked on all of the children in parallel. The parent creates the concurrency for these methods and controls it, waiting for the threads to finish and returning the results (an array of results, one element per child).

The Distributor pattern can also be used in other programs. It was used three times in the PSRS algorithm, and can be applied to any data–parallel problem.

After the design stage, the user selects the Distributor pattern template and instantiates the template. For the Distributor pattern template, the user specifies the names of the concrete classes for the parent and child classes (again, the abstract classes are automatically named) and a list with the following fields:

1. The name of the method that should be invoked concurrently on the child objects.
2. The return type for the child implementation of the method. The parent returns an array of this type, unless the type is `void`.
3. The arguments to the parent implementation of the method. For one–dimensional array parameters, the distribution of the parameter over the child objects must also be specified. Currently supported distributions are pass through, one element per child, striped distribution, block distribution, and neighbour distribution (child i gets a two element array of elements i and $i + 1$ from the original array). All other arguments are passed to the children directly (pass through distribution).

Note the third field of the tuples allows for one–dimensional array arguments to be automatically and correctly distributed to the child objects. For instance, we use the neighbour distribution in the first part of the fourth phase to distribute the partitioned elements to the children of the `Merger Container` class. We also distribute an array of indices, one per child, in the second part of the fourth phase so each child knows where it should merge its sorted partition. The code to distribute these arguments is part of the framework for the pattern and is not written by the user. The last parameter to the Distributor template, the number of children, is specified as a constructor argument in the generated framework.

Using the Framework Based on the specification of the pattern templates for this program, the structure of the framework for the PSRS program is given in Figure 8. The two uses of the Method Sequence framework are the `Sorter Sequence` and `Sorter Facade` class pair, and the `Merger Sequence` and `Merger Facade` pair. The two uses of the Distributor framework are the `Data Container` and `Data Child` pair, and the `Merger Container` and `Merger Child` pair. When generated, the framework does not contain the necessary references for composing the different frameworks; creating these references is covered in Section 3.2.5. However, any references needed in a particular framework are supplied in the generated code. For instance, the abstract sequence class has a reference to the Facade object in the Method Sequence framework. The actual object is supplied by the user to the constructor for the sequence class.

Both the Method Sequence and the Distributor frameworks have different hook methods that can be implemented. The sequence of method calls is shown in Figure 8. For the sequence object in the Method Sequence framework, these hook methods are:

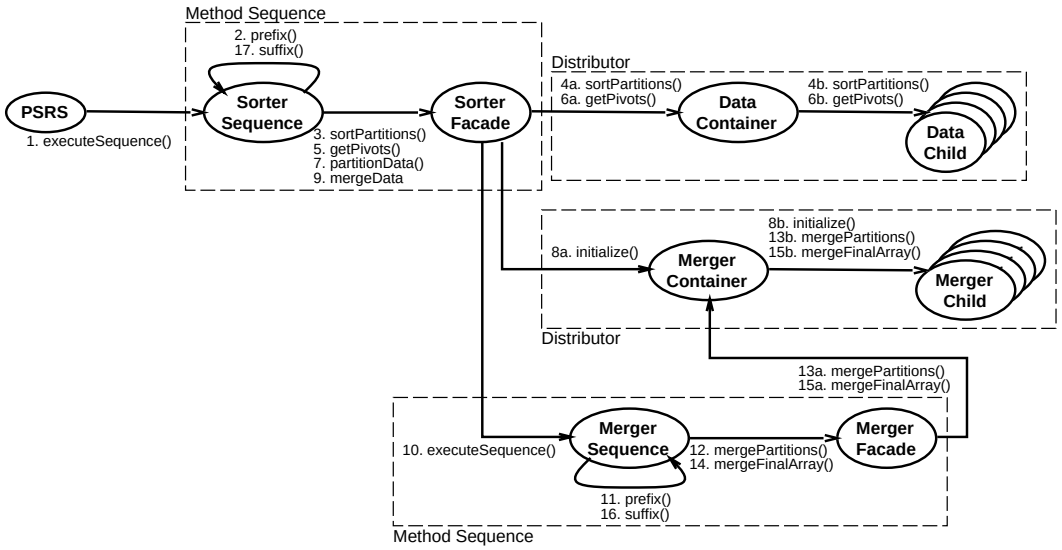


Figure 8: The structure of the PSRS program. The methods are executed in the order in which they are numbered.

`prefix()` This method is invoked before the methods in the sequence are executed. It can be used for any initialization required by either the sequence or Facade objects. For instance, in the `Sorter Sequence` class this method can be used to generate the data to be sorted.

`suffix()` This method is invoked after the methods in the sequence are executed. It can be used for any cleanup or postprocessing required by the program. For instance, in the `Sorter Sequence` class this method may verify the sort.

The hook methods for the Facade object are the sequence methods, the parameterless methods in the list of method names. These methods implement the different phases of the application. A phase has finished when its associated method returns, so all concurrent activity generating results used in another phase must be complete. Any partial results are stored in the Facade object.

For the Distributor framework, the hook methods are the child implementations of the methods specified in the last pattern template parameter. Each child operates independently, without reference to other child objects. These methods can operate on any state that has been distributed across the child objects or can be used to invoke methods on any distributed arguments in parallel. The parent object provides the structural code for this pattern, and has no hook methods. To assist the user, the signatures for the child methods are automatically generated and included in the concrete child class.

3.2.4 Evaluation

Since PSRS is a parallel algorithm, there is no sequential version. Therefore, we chose a sequential quicksort algorithm as a baseline for comparison. The sequential sorting program was 102 lines of Java code, used to initialize the data and verify the sort. The sorting algorithm was the quicksort method from the JGL library [8], which is 255 lines of Java code. The PSRS program, including the framework and collaborator classes, totaled 1252 lines of code (not including the JGL sorting code), 700 of which are user code. 414 lines of the user code are in the three classes `Data Child`, `Merger Child`, and `Data Container`. These classes contain most of the code for the application (the `Data Container` object is the single processor used for the second phase). Of the remaining classes, the two Facade classes and mainline are the largest. However, the methods in these classes consist mainly of accessors and, particularly in the two Facade objects, one line methods for delegating a method. The mainline also interprets command line arguments, and creates the `Sorter Sequence` object and the container for the data to be sorted.

In contrast to the reaction–diffusion example, the PSRS algorithm cannot make much use of the code from the sequential version. The problem is that the best parallel algorithm is not necessarily a parallel version of the best sequential algorithm. For instance, the performance of parallel quicksort peaks at a speedup of 5 to 6 regardless of the number of processes [12]. In these cases, writing the parallel algorithm requires more effort, as we see

with this problem. Nevertheless, the framework supplies about half of the code automatically, including the error–prone concurrency and synchronization code.

The performance results for PSRS, collected using the same environment given in Section 3.1.3, are shown in Table 2. These timings are only for sorting the data. Data initialization and sort verification are not included.

Unlike the reaction–diffusion example, both JIT and non–JIT versions of the PSRS program show good speedups, scaling to 16 processors. The principle reason for this improvement is that there are fewer synchronization points in PSRS; five for the entire program versus two per iteration of the mesh loop. In addition, the PSRS algorithm does more work between synchronization points, even with the smaller data set, reducing the overall cost of synchronization further.

From this example, we can see that CO_2P_3S also supports the development of explicitly parallel algorithms. The principle difficulty in implementing this kind of parallel algorithm is that little sequential code can be used in the parallel program, forcing the user to write more code (as we can see by the amount of user code needed for PSRS). Support for explicitly parallel algorithm development is crucial because a good parallel algorithm is not always derived from the best sequential algorithm.

3.2.5 Composition of Frameworks

Unlike the reaction–diffusion program, the PSRS example used multiple design pattern templates in its implementation, and required the resulting frameworks to be composed into a larger program. We explain briefly how this composition is accomplished, which also provides insights on how the user can augment the generated framework at the Patterns Layer.

In CO_2P_3S frameworks, composition is treated as it is in normal object–oriented programs, by delegating methods to collaborating objects. Note that the framework implementing a design pattern is still a group of objects providing an interface to achieve some task. For instance, in the code in Figure 6, the collector object provides an interface for the user to start the mesh computation and get the results, but the creation and control of the parallelism is hidden in the collector. If another framework has a reference to a collector object, it can use the Mesh framework as it would any other collaborating object, providing framework composition in a way compatible with object–oriented programming.

To compose frameworks in this fashion, the frameworks must be able to obtain references to other collaborating frameworks. This can be done in three ways: passing the references as arguments to a method (normal or hook) and caching the reference, instantiating the collaborating framework in the framework that requires it (in a method or constructor), or augmenting constructors with new arguments. The first two ways are fairly straightforward. The second method of obtaining a reference is used in the third phase of PSRS since the `Merger Container` object cannot be created until the pivots have been obtained from the second phase.

The third method of obtaining a reference, augmenting the constructor for a framework, requires more discussion as it is not always possible. We should first note that this option is open to users because the CO_2P_3S system requires the user to create some or all of the objects that make up a given framework (as shown in Figure 6). In general, users can augment the constructors of any objects they are responsible for creating. For the Mesh framework, the user can augment the constructor for the collector object. However, the added state can only be used to influence the parallel execution of a framework at the Patterns Layer if the class with the augmented constructor also has hook methods the user can implement. Otherwise, the user has no entry point to the structural code and the additional state cannot be used in the parallel portion of that framework. For instance, the user can augment the constructor for the parent object in a Distributor framework, but since the parent has no hook methods this state cannot influence the parallel behaviour of that object. However, new state can always be used in any additional sequential methods implemented in the framework.

4 Related Work

We examine work related to the pattern, pattern template, and framework aspects of the CO_2P_3S system.

Patterns There are too many concurrent design patterns to list them all. Two notable sources of these patterns are the ACE framework [10] and the concurrent design pattern book by Lea [5]. This work provides more patterns and attempts to provide a development system for pattern–based parallel programming. Specifically, our pattern templates and generated frameworks automate the use of a set of supported patterns.

Pattern Templates There are many graphical parallel programming systems, such as Enterprise [9, 13], DP-

Number of Elements	JIT disabled				JIT enabled			
	2 proc.	4 proc.	8 proc.	16 proc.	2 proc.	4 proc.	8 proc.	16 proc.
12,500,000	1.76 1519 sec	3.43 777 sec	6.74 395 sec	12.02 222 sec	1.73 567 sec	3.65 269 sec	7.09 138 sec	12.65 78 sec

Table 2: Speedups for the PSRS example. Wall clock times, rounded to the second, are also provided.

nDP [14, 13], Mentat [4], and HeNCE [1]. Enterprise provides a fixed set of templates (called *assets*) for the user, but requires the user to write code to correctly implement the chosen template, without checking for compliance. Further, applications are written in C, not an object–oriented language. Mentat and HeNCE do not use pattern templates, but rather depict programs visually as directed graphs, compromising correctness. DPnDP is similar to Mentat except that nodes in the graph may contain instances of design patterns communicating using explicit message passing. In addition, the system provides a method for adding new templates to the tool.

P^3L provides a language solution to pattern–based programming, providing a set of design patterns that can be composed to create larger programs. Communication is explicit and type–checked at compile–time. However, new languages impose a steep learning curve on new users. Also, the language is not object–oriented.

Frameworks Sefika *et al*. [11] have proposed a model for verifying that a program adheres to a specified design pattern based on a combination of static and dynamic program information. They also suggest the use of run–time assertions to ensure compliance. In contrast, we ensure adherence by generating the code for a pattern. We do not include assertions because we allow users to modify the generated frameworks at lower levels of abstraction. These modifications can be made to increase performance or to introduce a variation of a pattern template that is not available at the Patterns Layer.

In addition to verifying programs, Sefika *et al*. also suggest generating code for a pattern. Budinsky *et al*. [2] have implemented a Web–based system for generating code implementing the patterns from Gamma *et al*. [3]. The user downloads the code and modifies it for its intended application. Our system generates code that allows the user the opportunity to introduce application–specific functionality without allowing the structure of the framework to be modified until the performance–tuning stage of development. This allows us to enforce the parallel constraints of the selected pattern template.

Each of the PPSs mentioned above differ with respect to openness. Enterprise, Mentat, HeNCE, and P^3L fail to provide low–level performance tuning. However, Enterprise provides a complete set of development and debugging tools in its environment. DPnDP provides performance tuning capabilities by allowing the programmer to use the low–level libraries used in its implementation. Instead, we provide multiple abstractions for performance tuning, providing the low–level libraries only at the lowest level of abstraction.

5 Conclusions and Future Work

This paper presented some of the parallel design patterns and associated frameworks supported by the CO_2P_3S parallel programming system. We demonstrated the utility of these patterns and frameworks at the first layer of the CO_2P_3S programming model by showing how two applications, reaction–diffusion texture generation and parallel sorting by regular sampling, can be implemented. Further, we have shown that our frameworks can provide performance benefits.

We also introduced the concept of phased design patterns to express temporal relationships in parallel programs. These relationships may determine when to use parallelism and how that parallelism, if used, should be implemented. These phased patterns recognize that not every operation on an object has sufficient granularity to be run in parallel, and that a single parallel design pattern is often insufficient to efficiently parallelize an entire application. Instead, the parallel requirements of an application change as the application progresses. Phased patterns provide a mechanism for expressing this change.

Currently, we are prototyping the CO_2P_3S system in Java. We are also looking for other parallel design patterns that can be included in the system, such as divide–and–conquer and tree searching patterns. Once the programming system is complete, we will investigate allowing users to add support for their own parallel design patterns by including new pattern templates and frameworks in CO_2P_3S (as can be done in DPnDP [14]), creating a tool set to assist with debugging and tuning programs (such as the Enterprise environment [9]), and conducting usability studies [13, 15].

Acknowledgements

This research was supported by grants from the National Science and Engineering Research Council of Canada. We are indebted to Doug Lea for numerous comments and suggestions that significantly improved this paper.

References

[1] A. Beguelin, J. Dongarra, A. Giest, R. Manchek, and K. Moore. HeNCE: A heterogeneous network computing environment. Technical Report UT-CS-93-205, University of Tennessee, August 1993.

[2] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[4] A. Grimshaw. Easy to use object–oriented parallel programming with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[5] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley, 1997.

[6] S. MacDonald. Parallel object–oriented pattern catalogue. Available at http://www.cs.ualberta.ca/-∼stevem/publications.html, 1998.

[7] S. MacDonald, J. Schaeffer, and D. Szafron. Pattern–based object–oriented parallel programming. In *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 267–274. Springer–Verlag, 1997.

[8] ObjectSpace, Inc. *ObjectSpace JGL: The Generic Collection Library for Java, Version 3.0*, 1997. http://www.objectspace.com.

[9] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[10] D. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *Proceedings of the 12th Sun Users Group Conference*, 1994. A list of the individual patterns can be found at http://siesta.cs.wustl.edu/-∼schmidt/patterns-ace.html.

[11] M. Sefika, A. Sane, and R. Campbell. Monitoring compliance of a software system with its high–level design models. In *Proceedings of ICSE-18*, pages 387–396, 1996.

[12] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[13] A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.

[14] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 230–240, 1996.

[15] G. Wilson and H. Bal. Using the Cowichan problems to assess the usability of Orca. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.

[16] A. Witkin and M. Kass. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proccedings)*, 25(4):299–308, July 1991.