From Patterns to Frameworks to Parallel Programs

S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron,
K. Tan

Department of Computing Science, University of Alberta, Edmonton, Alberta, CANADA

Abstract

Object–oriented programming, design patterns, and frameworks are abstraction techniques that have been used to reduce the complexity of sequential programming. This paper describes our approach of applying these three techniques to the more difficult parallel programming domain. The Parallel Design Patterns (PDP) process, the basis of the CO_2P_3S parallel programming system, combines these techniques in a layered development model. The result is a new approach to parallel programming that addresses correctness and openness in a unique way. At the topmost development layer, a customized framework is generated from a design pattern specification of the parallel structure of the program. This framework encapsulates all of the structural details of the pattern, including communication and synchronization, to prevent programmer errors and ensure correctness. Lower layers are used only for performance tuning to make the code as efficient as necessary. This paper describes CO_2P_3S, based on the PDP process, and demonstrates it using an example application. We also provide results from a usability study of CO_2P_3S with real users.

Key words: Parallel programming systems. Design patterns. Object–oriented frameworks.

1 Introduction

Parallel programming offers the potential for substantial performance improvements to computationally–intensive problems found in fields such as computational biology, physics, chemistry and computer graphics. Solutions to these problems can take hours, days, or even weeks of processing time. However, to realize the potential, expert programmers must design highly concurrent algorithms that can execute on massively parallel computer systems. They must then implement these algorithms correctly and efficiently. This is a difficult and error–prone task that can benefit from the use of sophisticated parallel programming systems.

The solutions to many computationally intensive problems exhibit a degree of commonality that can be exploited. By extracting the common communication and synchronization patterns from these parallel solutions, we can build abstractions that

capture the expertise needed to write parallel programs. This idea is well known in sequential programming where such abstractions are called *design patterns* [8].

Even in the parallel domain, the idea of pattern abstraction is not new. For instance, a frequently recurring parallel solution is a grid or mesh [5,7]. A Mesh design pattern captures the experience of creating parallel programs for connected data elements, where each element iteratively computes a new value based on a combination of values in its immediate neighborhood. The elements are grouped into partitions, which are assigned to different processors and computed concurrently. Communication is required to exchange partition boundaries, which are needed to compute values for elements on the edge of a partition. Synchronization is needed to ensure that new values are computed with correct values. This basic strategy is common to all mesh algorithms and can be abstracted into a design pattern.

The Mesh pattern can be used to parallelize many finite element computations and image processing algorithms. Other parallel design patterns cover other problems that have traditionally been solved using task parallel, data parallel, or custom synchronization structures. However, design patterns capture design expertise at an abstract level. Patterns are applicable to different problem domains, each with different characteristics and concerns. A pattern is thus a family of solutions to a problem and must be adapted for each use. Once the programmer elects to use a pattern in a design, most of the basic structure of the program can be inferred. What happens next? Traditionally, the tedious process of implementing that pattern ensues. Sometimes old code is used to make the coding go faster. Unfortunately, old code is a combination of application–independent pattern structure and application–specific operations. The former code must be identified and extracted. Then the structure must be modified for its new use. This process can be time–consuming and error–prone because it can be difficult to recognize the application–independent code.

Fortunately, we can learn a valuable lesson about efficient code re–use from *frameworks* in sequential object–oriented programming [10,8]. A typical framework is a set of classes that implement the application–independent structure of a specific kind of program. This structural code defines the classes in the application and the flow of control between operations on the objects. This structure is written in terms of *hook methods* that are called when application–specific operations are needed. The hook method bodies are supplied by subclassing framework classes and overriding the methods. Using this style of programming, the design and implementation of the framework is used to write many different programs, reducing the effort needed to build applications [10]. Equally important are the organizational benefits of a framework. The application–independent structure is separate from the application–specific code and is never modified directly. This separation allows the user to concentrate on their application. It also reduces the probability of user error since the structure is not “contaminated” with application–specific code. It is important to recognize the difference between a framework and a library. With a library, the programmer writes the application structure and the library provides

application–independent utility routines. In contrast, a framework supplies the application structure and the programmer writes application–specific utility routines.

In this paper, we combine design patterns and frameworks in the parallel programming domain. In our approach, the user selects patterns for the program structure and a parallel programming system generates correct framework code. This code includes communication and synchronization. The hook methods are sequential code focused on solving the problem and not on the parallel structure. However, a pattern must still be adapted for its use. For this, we use *design pattern templates*, parameterized constructs based on design patterns. The parameters refine the pattern structure and guide the code generator. The resulting framework is specific to the parameters, improving its performance. pattern–based approach addresses correctness. Generating correct code for a parallel design pattern saves the user from writing and debugging this code, simplifying the creation of a parallel application.

Unfortunately, this pattern–based development model may not be sufficient to create high performance programs. The generated code may incur overhead that can be removed for a given program. Tuning the performance of a program requires that the programming system be *open*, allowing access to low–level features [23]. To address these concerns, we also include programming layers in our development model. The topmost layer supports pattern–based programming as described above. Lower layers gradually expose the generated code for performance tuning.

This combination of design patterns, frameworks, and layers forms the basis of the Parallel Design Patterns (PDP) process. This process is the philosophy underlying a new parallel programming system, CO_2P_3S (Correct Object–Oriented Pattern–based Parallel Programming System, pronounced "cops") [13–15]. As well as implementing the PDP process, CO_2P_3S provides tool support for program development with frameworks that enforces correctness. To demonstrate both the process and CO_2P_3S, we walk through the creation of an application. As well, we briefly discuss MetaCO_2P_3S, which introduces tool support for creating new pattern templates. We also present results from a study assessing the usability of the system.

2 The PDP Process for Pattern–based Parallel Programming

This section describes the PDP process, a pattern–based approach to parallel programming. It is based on a layered model that provides three distinct abstractions: the Patterns Layer, the Intermediate Code Layer, and the Native Code Layer. The Patterns Layer promotes correct parallel programming by generating frameworks for a set of supported patterns, and hiding the structure from the programmer. The Intermediate Code Layer and the Native Code Layer gradually expose the details of the frameworks. The framework can then be tuned to remove performance bottlenecks or modified to implement a pattern variation.

The process has five steps. The first three are required and the last two are optional:

(1) Identify one or more parallel design patterns that can contribute to a solution and pick the corresponding design pattern templates.
(2) Provide application–specific parameters for the design pattern templates to generate collaborating framework code for the selected pattern templates.
(3) Provide application–specific sequential hook methods and other non–parallel code to build a parallel application. The Patterns Layer is now complete.
(4) Monitor the parallel performance of the application and if it is not satisfactory, inspect the generated parallel–structure code at the Intermediate Code Layer. This code contains high–level synchronization and communication primitives. Remove primitives that are not necessary in the specific application or modify the location and parameters of the primitives to improve performance.
(5) Re–monitor the parallel performance of the application and if it is still not acceptable, modify the implementation of these primitives for your specialized target architecture at the Native Code Layer.

We illustrate the PDP process and the application architecture by using an example problem solved with a Mesh design pattern. The process is illustrated using our programming environment called CO_2P_3S which generates shared memory Java code. CO_2P_3S should be viewed as an example of the process. Other environments can be created that support different programming languages, different patterns, or require different pattern template parameters and hook methods. This paper focuses on steps two and three in the process and only briefly describes the other steps.

2.1 Selecting a Design Pattern

A parallel design pattern is the encapsulation of a strategy for solving a problem using a familiar parallel communication strategy. Given a palette of design patterns, the programmer must select the appropriate patterns for their application. We do not address this selection process. It is possible for programmers to select a pattern template that is inappropriate for the problem. However, there are several methodologies for finding the best parallel structure for an application [5,7,16].

For clarity, we have chosen an example problem that can be solved with a single pattern. It is often the case that several patterns must be combined in a solution. Regardless, the patterns must be documented to facilitate the pattern selection process. We enhanced the standard pattern description in [8] to include parallel aspects.

Consider a reaction–diffusion simulation of chemicals, called *morphogens*, over a two–dimensional surface [30]. The simulation results can be used to produce a zebra stripe texture. We would like to tile this texture over a large surface without discernible edges. The simulation uses two morphogens, starting with random concentrations and the correct reaction and diffusion parameters. This problem is

similar to solving two interacting Laplace equations. We solve it using convolution.

The first step of our process is to pick a design pattern for solving the problem. We can consider the Mesh design pattern. Specifically, we consider a Two-Dimensional Regular Mesh pattern. For the rest of this paper, the Mesh pattern refers to this pattern, and we will use mesh computation to refer to this particular computation.

The Mesh design pattern supports computations on data in a rectangular data structure, where each element computes its new value using its current value and values from the elements around it. This computation is executed iteratively, usually until the elements converge to a solution. Mesh computations are parallelized by splitting the data into partitions and assigning each to a different processor. However, the partitions contain a data dependency, as the elements on the edge of one partition require elements from adjacent partitions to compute new values. This dependency defines the synchronization and communication structure of a parallel mesh.

Within this pattern, there are several options to consider. These options affect the implementation of the Mesh pattern. For example, a mesh can use synchronous or asynchronous Jacobi iteration. Another option is how to handle the edges of the data, or the mesh *topology*. The edges may wrap in one or more directions.

The reaction–diffusion example computes new values using finite differences, determining the change in concentration values in the morphogens based on the concentrations at neighboring cells. The problem uses synchronous Jacobi iteration to iteratively compute the new values until the morphogen concentrations converge, as the problem specification indicates the values from the previous iteration be used. To generate a texture that can be tiled, the morphogens diffuse around the edges in both the horizontal and vertical directions. The Mesh pattern is appropriate for this problem, so we select the Two–Dimensional Regular Mesh design pattern template.

2.2 From Design Patterns to Pattern Templates

To incorporate the idea of patterns as families of solutions, we introduce a new concept called a *design pattern template*. This template represents the basic structure of the pattern but includes parameters to specialize this structure to include common alternatives. Thus, pattern templates are a parametrically related family of solutions, much like the patterns they are based upon. Using a template, a programmer can select the pattern structure that best matches their application. For example, the parameters for a Two–Dimensional Regular Mesh template include the topology, the number of neighboring elements, and the synchronization level.

We inspect the documentation of the Mesh design pattern template to see what template parameters we must provide. In CO_2P_3S , there are five parameters:

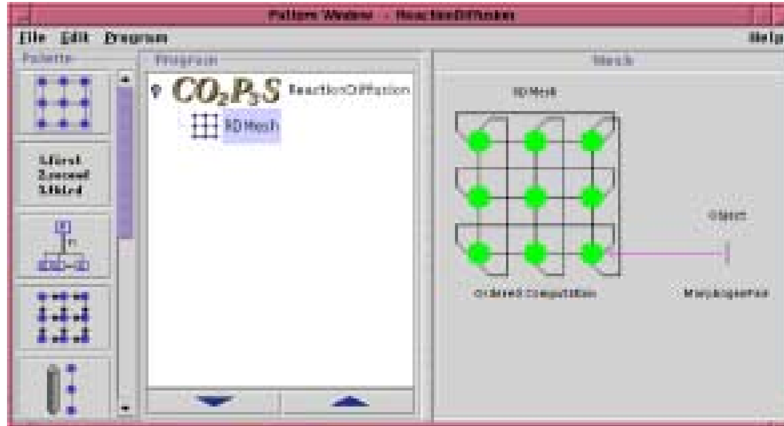

Fig. 1. The reaction–diffusion example in CO_2P_3S, with the Mesh pattern template.

(1) The name of the class representing the mesh. We use `RDMesh` in our example.
(2) The class name for the elements that populate the two–dimensional data structure. We use `MorphogenPair`, a class that holds a pair of morphogens for each cell on the surface. This class has no application–specific superclass.
(3) The topology of the mesh. This application uses a fully–toroidal mesh to allow the morphogens to diffuse around the edges of the surface. The resulting texture can be tiled on a larger display without any noticeable edges. The template also supports non–toroidal, horizontal–toroidal, and vertical–toroidal meshes.
(4) The pattern of neighboring elements that each element uses to compute a new value. In this program, the morphogens diffuse up–and–down and side–to–side, so we use a four–point mesh. An eight–point mesh is another option.
(5) The synchronization level: whether the iterations are synchronous or asynchronous. With synchronous iteration, each element waits for its neighbors to finish computing their current values before computing the next value. With asynchronous iteration, each element computes its next value as quickly as it can. This problem uses synchronous iteration.

Figure 1 shows an example of how the template parameters for the reaction–diffusion program might be selected and specified in a programming environment. It is a screenshot of the CO_2P_3S user interface showing a Mesh pattern template.

When designing pattern templates, we must balance usefulness against generality. We can add more parameters to a template to provide a larger number of alternative structures. However, a larger number of parameters can make the specification of a template more complicated. On the other hand, fewer parameters may mean that either some aspects of the basic structure cannot be specified using the template, or that the generated code is so general that it is inefficient. Striking this balance requires us to consider the alternatives on a pattern–by–pattern basis. Consider the parameters in the Mesh template as an example. The topology and neighbor parameters have the greatest effect on the generated framework code.

Consider the mesh topology parameter. Ultimately, a mesh computation requires

each element to update itself based on the values of its neighbors. This computation is performed by a sequential hook method that is automatically generated from the pattern template and implemented by the user. Its method header is of the form:

```
void operate(MeshElem right, MeshElem left,
             MeshElem up, MeshElem down);
```

However, if a mesh element is on a boundary it has fewer neighbors. There are two solutions to this problem. The first is to require the user to implement this single method by checking the location of the element and computing its value with the available neighbors. A better solution is to generate different methods for each boundary condition, with appropriate arguments. The framework invokes the correct method with the proper neighbors for each mesh element. This is an example of error–reduction due to code generation that is discussed later. In general, there are nine possible location–dependent methods, listed in [13]. However, only a subset of these are applicable for a given topology. For example, a fully–toroidal mesh has no edges or corners. The generated mesh element class only includes stubs for the operations that are needed. Generating methods that are not used inflates code size and can be confusing to a user who sees hook methods that are not called. This is a good example of how the template parameters guide the code generation process.

Now consider the neighbor parameter for the mesh. This parameter determines the arguments for the generated operation methods. A four–point mesh considers the elements at the compass points, where an eight–point mesh also includes diagonal elements. Without this mesh parameter, a choice would have to be made to always generate four or eight argument methods. Using four arguments reduces generality. Using eight arguments adds memory accesses if only four arguments are needed. Once again, parameterized pattern templates provide a solution to this problem.

2.3 From Pattern Templates to Frameworks: The Patterns Layer

Each design pattern template represents a parametrically related family of pattern implementations. After the user specifies the template parameters, a code generator produces a framework for the pattern and its parameters. This framework consists of abstract classes implementing the pattern structure, including concurrency, synchronization, and communication. The user does not need to modify this code.

The PDP process is independent of programming language and parallel architecture. CO_2P_3S, one implementation of this process, generates multithreaded Java framework code for shared memory multiprocessor systems. Some of our current research is aimed at creating distributed memory pattern templates.

A set of concrete classes is also generated. The sequential hook methods in these concrete classes are invoked by the parallel structure code at the appropriate time. In step three of the PDP process, the programmer must implement these sequen-

tial hook methods to implement the application. In addition, the programmer must provide code to create the appropriate application objects, to start the parallel computation and (possibly) to gather the results of the parallel computation. The generated framework code, together with the hook method code, the initialization code and the result–gathering code are called the *Patterns Layer Code*.

Note that the generated frameworks are not targeted at a particular problem domain. The frameworks provide code implementing the specific pattern indicated by the template parameters so that programs from different domains can be implemented.

An alternative to generating a new framework for each design pattern template instance is to create a single framework that the user can instantiate for each use in a program. However, there are several benefits to generating code for each pattern template instance. One benefit is that the framework can be customized for the selected structure. This can improve performance by reducing indirection in the code, which would be necessary to accommodate structural variations of the pattern in a generic implementation. Another benefit is that it is difficult to incorporate pattern variations that involve the use of application–specific interfaces or the redistribution of responsibilities between objects into a single framework. A final benefit is that we are not limited to generating only framework code for the pattern templates. We can also generate additional support code to simplify the instantiation and use of the frameworks. For instance, we generate concrete classes with stubs for the hook methods with the framework code. Other examples of support code, to insulate the user from implementation details of the framework, will be discussed later.

Before using the framework generated for the Mesh design pattern template to implement our example application, we need to consider what parts of the program are provided by the framework and what parts need to be implemented by the programmer. The programmer is responsible for implementing a class representing an individual mesh element using the operations defined by the hook methods. In the reaction–diffusion example, this class would correspond to a pixel in the texture.

The structural framework code builds up a complete parallel mesh computation based on the individual mesh elements. The structure populates the mesh with the user–defined elements, creates the threads, partitions the elements over the set of threads, executes the computation, and gathers the results. The heart of the mesh computation, the main execution loop, is shown in Figure 2. This method is executed by each thread in the computation on its local partition. Since this method is part of the parallel structure, the programmer cannot modify it. In fact the programmer does not even have to see it. However, it is useful to discuss it to understand how the parallel mesh structure works. In general, each of the methods in Figure 2 iterate over the local partition, invoking a similarly named method on each element.

The `meshMethod()` method repeatedly executes the mesh computation on the elements in a local partition until the termination condition is satisfied. `notDone()`

```
public void meshMethod() {
    preProcess() ;
    while(notDone()) {
        prepare() ;
        barrier() ;
        operate() ;
    } /* while */
    postProcess() ;
} /* meshMethod */
```

Fig. 2. The main execution loop for the Mesh framework.

checks the termination condition for the mesh. The computation continues until all mesh elements are finished. `preProcess()`, `prepare()`, and `postProcess()` invoke similar methods on each mesh element. `operate()` is also mapped to each mesh element, resulting in one of nine appropriate operation methods being invoked on the element. `barrier()` does not map directly to each mesh element. It ensures that the current mesh operation does not proceed until the previous ones have completed. Although the code in `meshMethod()` does not reveal how the mesh is partitioned into groups and how the groups are mapped to processors, these details are not germane to this paper. More importantly, they are not germane to the programmer's task (at this layer of abstraction) and they would only serve as a distraction to the task of implementing the application code.

The Mesh framework includes a main class that creates the mesh and launches the computation. The constructor takes the size of the surface, the number of horizontal and vertical partitions, an initializer object, and a reducer object as arguments. When the computation is launched, each thread starts executing `meshMethod()`. When the computation is finished, the results are gathered and returned to the user.

The only class that the programmer implements is the mesh element class. There are five tasks in total. The first is to write a constructor that applies a user–defined initializer object to the mesh element being created. The second is to write the basic mesh operations for the chosen topology for a single element. The third is to write the three other methods for each element. The fourth is to implement the termination condition for an element. The fifth is to write a method that applies a user–defined reducer object to a mesh element. These methods are listed in Table 1.

In the example program, the mesh element class is `MorphogenPair`. Complete code for this class is in [13]. The initializer object is a random number generator, so each mesh element can create two morphogens with random initial concentrations. `notDone()` checks for convergence. `interiorNode()` computes the new value for each morphogen based on the four neighbors supplied by the framework. The synchronous Jacobi iteration is implemented by having each morphogen hold two concentrations, a write value for the value computed by the current iteration and a read value used to compute new values. `prepare()` updates the read

Table 1
The hook methods for the mesh element class in a four–point mesh.

Hook method with signature	Implemented responsibility
`MeshElem(int i, int j, int surfaceWidth, int surfaceHeight, Object initializer);`	This method initializes the mesh element at the location (i, j) of the two–dimensional structure, by applying the user–supplied initializer object.
`void preProcess();` `void prepare();` `void postProcess();`	These methods allow application code to be inserted at various points in the mesh computation. The `barrier()` call in Figure 2 implements the necessary synchronization in the mesh structure. It ensures that all of the threads have finished any preprocessing for an iteration (the `prepare()` call) before they compute the next value for the mesh elements.
`boolean notDone();`	This method is called indirectly from the `notDone()` method in Figure 2. It evaluates the termination condition for a single mesh element. The computation continues until all mesh elements return `false`.
`void reduce(int i, int j, int surfaceWidth, int surfaceHeight, Object reducer);`	This method is responsible for applying the programmer–supplied reducer object to gather the results of the mesh computation after it is done.

value with the write value. The barrier synchronization ensures that this update is complete before new values are computed. Finally, `reduce()` applies a reducer to each mesh element to gather the results. This program copies the concentration of one of the morphogens into an output array. Additional instance variables, accessor methods, and constants for the simulation are also included in this class.

The code for this problem is written using several other classes that were taken directly from a sequential implementation of this problem. The hook methods are used as Adapters [8], translating the interface to the hook methods to the interface supported by the sequential code. It is also possible to use the hook methods to wrap code libraries written in different languages. The Java Native Interface allows Java code to invoke C functions, which can invoke C++ or FORTRAN code.

Performance is the most vital characteristic of parallel programming systems since improved performance is the reason for writing parallel programs. To show that CO_2P_3S can generate applications with reasonable parallel performance, even without tuning, performance results for the reaction–diffusion example are shown in Table 2. These performance numbers are not necessarily the best that can be achieved, but show that it is possible to build a parallel program and quickly obtain reasonable speedups. The program was run using a native threaded Java implementation

Table 2
Speedups and wall clock times for the reaction–diffusion example.

		2 proc.	4 proc.	8 proc.	16 proc.
1680 × 1680	Speedup	1.75	3.13	4.92	6.50
surface	Time (sec)	5734	3008	1910	1448

with optimizations on. The program was run on an SGI Origin 2000 with 195MHz R10000 processors. The virtual machine was started with 512MB of heap space. The speedups are based on wall clock times compared to a sequential program.

The speedups only include computation time; neither initialization nor output is included. From Table 2, we can see that the problem scales well up to 4 processors, but the performance falls off afterwards. The problem is granularity; as more processors are added, the amount of work assigned to each falls until synchronization becomes a limiting factor in performance. Larger computations (either with a larger surface size or with a more complex mesh operation) yield better speedups.

Other example programs, implemented with other pattern templates, include Parallel Sorting by Regular Sampling [13], 15–puzzle search [13], JPEG encoding [13], sequence alignment [1], matrix product chain [1], and a skyline matrix solver [1].

The two crucial aspects of the Patterns Layer are parallel structure correctness and parallel separation. *Parallel structure correctness* guarantees that there are no communication, synchronization, or parallel access errors in the structural part of the framework. This is due to the fact that the user only edits sequential hook methods and cannot affect th parallel code. Only the structural code access the data structures internal to a given framework. The hook methods receive this data as arguments, such as the neighboring element parameters in the Mesh framework.

It is vital to note that this correctness guarantee does not extend to the application–specific code written by the user. It is possible for hook method code to introduce errors, such as unprotected accesses to a static variable. The user must ensure that such accesses do not occur. Unfortunately, code libraries can make it difficult to find such problems. Barring changes to the programming language, there is no general solution to this problem. We do not address it in this research.

Parallel separation means that the parallel structure code is separated from the application–specific code. The generated code and the application–specific code are in different classes This can limit the number of changes that are necessary in the hook methods if the framework code is changed. In many cases, the user can change template parameter values and regenerate framework code with minimal changes to the hook methods. For example, if the programmer changes the mesh topology from fully–toroidal to horizontal–toroidal, the implemented hook methods do not need to be changed. The programmer need only write hook methods for the new boundary conditions. Stubs for these new methods are generated with the framework.

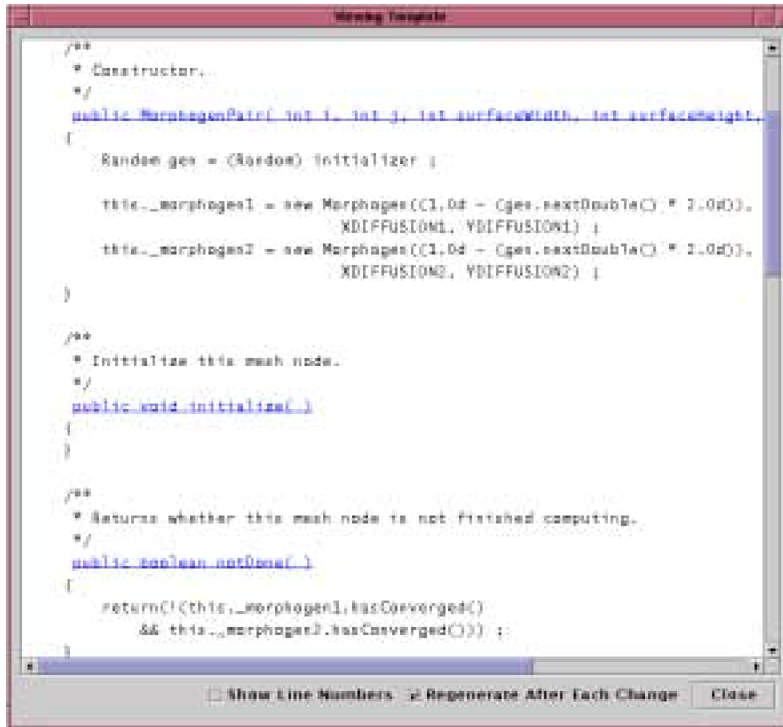

Fig. 3. A modified HTML viewer for entering hook method code.

This separation can be further enforced by tool support. In CO_2P_3S, the application–specific code is presented in a modified HTML viewer shown in Figure 3. This viewer only allows the user to access the hook method bodies. When a link is selected, a dialog for entering code appears that does not permit the hook method signature to be changed, removing a source of potential programmer errors.

2.4 From Frameworks to Parallel Programs: The Lower Layers

Once the hook methods in the generated framework have been implemented, the programmer has a fully functioning, structurally correct parallel program that can be executed on a parallel machine. Ideally, the Patterns Layer would be sufficient to create an efficient parallel program. Unfortunately, this is not possible for two reasons. First, it would require the set of available pattern templates to cover the complete spectrum of parallel program structures, which is not possible. Second, even though the pattern templates can be specialized by the user, the generated frameworks are conservatively correct. They are general enough that they can be used for a variety of problem domains. Consequently, some portions of the framework may not be optimal for a specific problem. The principle purpose of writing parallel programs is to improve the performance of an application, so we need a way of tuning the generated code to remove performance problems. Most parallel programming systems do not have any means of accessing, much less tuning, code inserted by the system, a major shortcoming that is addressed by our research.

To support application tuning, the PDP process has two additional layers. These layers provide openness by gradually exposing the structural code encapsulated at the Patterns Layer. The second layer, the Intermediate Code Layer, provides a high–level explicitly parallel object–oriented programming language, an extension of an existing language such as Java. This layer includes high–level primitives such as barrier synchronization. Also, the structural code is made available. The programmer can change or augment this code (to parallelize the `reduce()` method, for example). The programmer can also write applications in this language, bypassing the Patterns Layer. The implementation of the new primitives at this layer is hidden.

If the performance of the application is still not acceptable after tuning the structure at the Intermediate Code Layer, it is still possible to improve performance by tuning the implementations of the primitives at the Native Code Layer. This layer provides access to the programming language and libraries that underly the higher–level abstractions. The Native Code Layer contains the architecture–dependent aspects of the system, which can be tuned for the execution environment. For example, the barrier operation can be tuned for the particular computer used to run the program.

While the lower layers provide the openness needed for performance tuning and other program modifications, they also remove the correctness benefits from the Patterns Layer. Once the user modifies the code, structural correctness cannot be guaranteed. With CO_2P_3S, it is always possible to regenerate the Patterns Layer Code and start tuning again with a correct program if errors are introduced at the lower layers. An interesting problem is how to take changes at the lower layers and incorporate them into the pattern template. Our tool support for extending CO_2P_3S, with new or modified pattern templates, is discussed briefly in Section 3.3.

Dividing the performance tuning steps into multiple layers offers usability benefits. By gradually exposing the framework details, users can select a suitable abstraction based on the problem being solved. For instance, adding code to count the number of iterations in a mesh computation should not require the user to consider the implementation of the barrier synchronization. Also, users can select an abstraction based on how comfortable they are with it, but can still expect tuning opportunities. Thus, we believe that the multiple layers of abstraction provided by the PDP process will benefit novice parallel programmers while providing low–level control for experienced programmers. This support for multiple layers of abstraction to tune and refine parallel programs is a further advance in parallel programming systems research that creates a flexible environment for building efficient parallel programs.

3 The Usability of CO_2P_3S

One of the most important but least studied problems in parallel programming systems is usability. Every parallel programming tool claims to be easy to use based on

anecdotal evidence from its developers. The possibility of inadvertent bias cannot be discounted. The developers of a system will have a better understanding of the tool, which is a large advantage over normal users. Ultimately, the tools we build are intended to be used by other developers. We will require feedback from these users if we hope to build parallel programming systems that they will use. This section describes results from the few available usability studies. The results of another study assessing the impact of pattern–based parallel programming systems on program complexity and maintainability is also discussed. Finally, the results of an initial usability study on CO_2P_3S are presented.

3.1 The Usability of Enterprise and Orca

The first study compared parallel programming with the pattern–based Enterprise system against the PVM [26]. Users identified three strengths of Enterprise. First, the system prevented several common parallel programming errors. Most errors in Enterprise affected application performance and not correctness. Second, programmers were able to quickly create a working parallel program. Finally, Enterprise had good tool support. In particular, many users took advantage of a replay mechanism to visualize an execution of their program to identify performance problems. The study also uncovered three weaknesses in Enterprise. First, programmers could quickly create a working program but were unable to tune its performance. Second, the programming model of Enterprise introduced subtle changes to the programming language semantics. Confusion over these new semantics was the primary source of performance errors. Last, Enterprise users found that the performance of their programs was poorer than their PVM counterparts. In combination with their inability to tune performance, this problem frustrated many Enterprise users.

CO_2P_3S maintains the first two strengths by generating correct framework code from the pattern templates. This saves users from having to write complex parallel code at the Patterns Layer. Encapsulating the structural code reduces the possibility of user errors. The user can concentrate on the application code rather than the framework. Although frameworks impose their own learning curve [10], starting program development with a correct structure can only improve the usability of the system. Development time is reduced because it takes little time to select a pattern template, supply parameter values, and generate the code. It can be reduced further by using sequential code in a parallel application. At present, CO_2P_3S does not have an integrated set of support tools. This will be a subject of new research.

The PDP process addresses the first and third problems through the layered programming model. The layers provide successively lower–level details and control over the implementation of the generated frameworks, providing programmers with increased control and opportunities for performance tuning. However, the second problem is also present in the Intermediate Code Layer of the PDP process to sup-

port the development of highly concurrent algorithms. However, unlike other systems, the PDP process relies on frameworks and libraries to express concurrency. This limits the number of semantic changes. However, other problems remain. For instance, CO_2P_3S currently does not have good support for safely accessing global data. Such support will need to be added to reduce potential programmer confusion.

The study for the Orca parallel programming language [29] is based on user experiences with the Cowichan problem set [28]. The Cowichan problems are a set of seven modestly–sized programs that cover a broad range of application domains and parallel programming idioms. The primary lessons of the study are two–fold. First, tools are essential to identifying performance problems. Second, the programming model for a parallel language should be general. Orca suffered from two serious limitations that required significant programming effort to overcome.

Future work on CO_2P_3S will concentrate on tool support. The generality of any system implementing the PDP process is a function of the pattern templates it supports. The extensibility and openness of a system will dictate the ability of users to address any limitations that they encounter. In CO_2P_3S, users can modify a generated framework at lower development layers. If such changes are made frequently, they can be added as a new pattern template or a parameter to an existing template.

Another study assessed the impact of pattern–based systems on application complexity and maintainability [27]. The data for this study was taken from programs written with MPI [25], Enterprise, FrameWorks (not to be confused with object–oriented frameworks) [22], and PAS [9]. An analysis of the code revealed that programs written with pattern–based systems were less complex and more maintainable than the MPI equivalents.

3.2 A Usability Study of CO_2P_3S

We have undertaken a usability study of CO_2P_3S that is similar to the study for the Enterprise system. This study was conducted over two assignments in an undergraduate course with 20 students. The first assignment had the students solve a modified version of the Laplace equation, and the second was the reaction–diffusion problem. The class was split into two groups of 10 students. The first group wrote the first assignment using non–CO_2P_3S Java (with a barrier class provided) and the other group used CO_2P_3S. The groups switched development systems for the second assignment. Our results focus on the code written for the assignments as a measure of the difficulty of writing the two programs. Other measures, such as the number of compiles, program runs, and development time, are not considered here. Tables 3 and 4 summarize the results. In both assignments, the CO_2P_3S group wrote less code, 40% for the first assignment and 53% for the second. The CO_2P_3S students also used fewer classes, 41% in the first assignment

Table 3
Results from the usability study for CO_2P_3S, for the Laplace problem.

CO_2P_3S students				Non–CO_2P_3S Java Students			
No. Programs	Avg. Lines of Code	Avg. No. Classes	Avg. No. Choice Points	No. Programs	Avg. Lines of Code	Avg. No. Classes	Avg. No. Choice Points
10	171.6	4.1	20	8	274.9	6.6	52.6

Table 4
Results from the usability study for CO_2P_3S, for the reaction–diffusion problem.

CO_2P_3S students				Non–CO_2P_3S Java Students			
No. Programs	Avg. Lines of Code	Avg. No. Classes	Avg. No. Choice Points	No. Programs	Avg. Lines of Code	Avg. No. Classes	Avg. No. Choice Points
6	131.0	4.2	11.8	6	278.5	6	46.5

and 30% for the second. The difference is the generated structural code, which the non–CO_2P_3S students had to write by hand. To assess complexity, we measure the number of choice points in the application code. Choice points are any point in a program where the flow of control can be altered and may no longer be sequential, such as selection control statements. Errors in programs tend to occur at these choice points when the user makes the wrong choice. We can see that the application code of CO_2P_3S users is significantly less complex than that written by their counterparts. Most of the complex code in a mesh computation is in the structure, which is generated for CO_2P_3S users. Note that choice points do not consider the effort required to determine where the application code should be inserted within the hook methods in a framework. This is an important aspect of program development with a framework that can be improved only through experience [10].

3.3 Tools for CO_2P_3S Extensibility

Having realized that the need for tools to enable CO_2P_3S extensibility was one of our most pressing concerns to improve system usability, we have concentrated our latest research efforts in this area. MetaCO_2P_3S will enable a pattern designer to create a pattern template/framework pairing that integrates seamlessly with CO_2P_3S [3]. The pattern templates and frameworks created using the tool are indistinguishable in both looks and functionality from the ones provided. These new templates can be shared amongst CO_2P_3S users. In the future, we envision a pattern repository that will allow a community of users to freely exchange their pattern templates.

With MetaCO_2P_3S, a pattern designer has three main responsibilities. First, they must define the pattern template, including the parameters that can act upon it and

its GUI representation. This information is stored in a system–independent XML file that is imported into CO_2P_3S. The pattern designer's second responsibility is to define the framework. Since the framework code is parameterizable, this task is not trivial. Our approach has the pattern designer create annotated source code under the guidance of the tool. When a user generates a framework in CO_2P_3S, we run the annotated source code through a modified JavaDoc parser and output the appropriate code. The last responsibility of the pattern designer is to ensure that the frameworks for the template are correct. We verify our frameworks by using them to write applications after the design of a pattern template is complete.

To demonstrate the usefulness of MetaCO_2P_3S, all of the design pattern templates currently supported by CO_2P_3S have been generated using it. In addition, a new pattern template, the Wavefront, has been created and used to implement three different problems [1], including two from the Cowichan problem set [28].

Note that pattern designers are not limited to creating new pattern templates. It is also possible to modify existing templates, adding new parameters and generating different framework code. Templates can also be copied and ported to different parallel architectures. We are porting our existing templates to distributed memory machines, using the shared memory versions as a starting point.

Additional information on MetaCO_2P_3S can be found in [3].

4 Related Work

A number of research groups have developed pattern–based parallel programming tools. FrameWorks was one of the first [22]. FrameWorks programs consisted of a combination of annotated source code and structural diagrams describing the parallelism. However, it was the responsibility of the user to ensure that both of these parts of the program were in agreement. In addition, the patterns supported by FrameWorks were not independent; some combinations did not work properly.

Enterprise improved on FrameWorks by transforming sequential C code based on a structural diagram [18]. Further, the patterns in Enterprise were independent and easily composed. However, like FrameWorks, Enterprise did not take full advantage of the asset graph, using it to verify the program structure rather than create it.

Like our work, DPnDP uses the design pattern information to generate code for the pattern [24]. Using the pattern description, all pattern–specific communication is handled automatically. The programmer explicitly exchanges application–specific messages. The system addresses extensibility, allowing users to add new patterns. However, the support for extensibility is not complete. New patterns are added using C++ framework rather than using tools. The user interface of DPnDP cannot be

easily extended to include the new pattern, whereas part of the templates created with MetaCO$_2$P$_3$S include GUI information. Also, only the structure of new patterns can be added. Behavioural aspects, such as pattern–specific communication, cannot be added. It is not possible to recreate the patterns supplied with DPnDP.

PAS provides a number of communications interfaces tailored for specific pattern structures, such as workpiles and meshes [9]. Applications are written either using a specification language that includes special constructs for pattern information or using templated C++ code and instantiating the correct communication interface. Regardless, the application code includes communication code using the appropriate interface. New communication interfaces can be added, but it is unclear if the specification language can be easily extended to incorporate them.

PAS is based on algorithmic skeletons [6]. Skeletons are another means of expressing parallel structures that can be used be build applications. A skeleton can be likened to a single framework that encompasses all possible structural variations.

The CORRELATE language [17] also relies on code generation to support its used of the Active Object pattern [11]. Remote method invocations are annotated, and a precompiler processes the annotations to generate all of the classes needed to implement the pattern. However, this support is only targeted to a single pattern.

P^{3}L is a language–based solution to pattern–based programming [2]. A P^{3}L program is a set of code fragments and a pattern description that describes how the fragments are composed into a complete program. The program is then mapped onto the underlying hardware architecture by creating an *abstract machine* tailored for the pattern and execution environment. Communication in a program is handled by typed input and output parameter streams rather than message passing primitives, but is still addressed in application code. The language does not appear extensible, as this would require a pattern designer to create new abstract machines.

There has been considerable work on concurrent design patterns, such as that of Lea [12], the ACE project [19], and POSA2 [20]. While most of this work does not address the performance aspects of parallel programming, concurrency is a crucial component of parallel algorithms. Further, the concurrent structural patterns in most patterns literature consists of very fine–grained patterns solving small, specific problems, whereas this work concentrates on larger structures.

Common parallel structures have been known for some time now and are described in introductory parallel design and algorithm literature [5,7]. However, the pattern–based systems discussed earlier are representative of the research in tools that support program development using these patterns.

The idea of generating code for patterns is not new. Budinsky *et al.* [4] developed a web–based tool that creates code for the patterns in [8], including the ability to select pattern variations from the options in the pattern documentation. However,

the generated structural code is returned to the programmer, who can introduce structural errors. We follow a similar strategy, but hide the generated code until performance tuning is required. Sefika *et al.* [21] also suggest generating structural code for patterns, and also advocate adding run–time assertions to ensure a system adheres to its design. We do not include assertions in our patterns as our open programming model is designed to allow the structure of a pattern to be modified.

5 Conclusions

In this paper we presented the PDP process, a pattern–based process for building parallel programs that addresses correctness and openness. This new process uses a layered approach to creating applications. The first layer emphasizes the correctness of programs, and is based on generating structural framework code from a pattern description. Lower layers provide openness, allowing programmers to access the framework code for performance tuning. It is this layered approach that addresses many of the shortcomings of existing systems. Finally, we used CO_2P_3S, a tool that implements the PDP process, to show that we can use this process to build correct, working parallel programs that yield performance improvements.

Although we have addressed the issues in pattern–based programming systems, we have not fully assessed the usability of our process or our tool. We have some results and feedback that we can use to make improvements in the tool and the process. Further work should be undertaken to assess the usability of our frameworks, which is essential to the success of any tool implementing the PDP process. We can model these usability experiments after others for parallel systems [26,29].

Current research is concentrating on finding new patterns to add to CO_2P_3S, and porting the system to distributed memory architectures. Future work will address supporting tools to produce a fully–featured, mature parallel programming system.

An initial version of CO_2P_3S is available for download. It can be found at
`http://www.cs.ualberta.ca/~systems/cops/index.html`

Acknowledgments

This work was supported by the Alberta Research Council, the Natural Science and Engineering Research Council of Canada, Alberta's Informatics Circle of Research Excellence, and MACI (Multi–media Advanced Computational Infrastructure).

References

[1] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating parallel programs from the wavefront design pattern. In *Proceedings of 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2002.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[3] S. Bromling. Meta–programming with parallel design patterns. Master's thesis, Department of Computing Science, University of Alberta, 2001.

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

[5] K. Mani Chandy and S. Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett Publishers, 1992.

[6] M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. MIT Press, 1988.

[7] I. Foster. *Designing and Building Parallel Programs*. Addison–Wesley, 1995.

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

[9] D. Goswami, A. Singh, and B. Priess. Using object–oriented techniques for realizing parallel architectural skeletons. In *Proceedings of the Third International Scientific Computing in Object-Oriented Parallel Environments Conference*, volume 1732 of *Lecture Notes in Computer Science*, pages 130–141. Springer–Verlag, 1999.

[10] R. Johnson. Frameworks = (components + patterns). *CACM*, 40(10):39–42, 1997.

[11] R. Lavender and D. Schmidt. Active object: An object behavioral pattern for concurrent programming. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*, chapter 30, pages 483–499. Addison–Wesley, 1996.

[12] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley, second edition, 1999.

[13] S. MacDonald. *From Patterns to Frameworks to Parallel Programs*. PhD thesis, Department of Computing Science, University of Alberta, 2002.

[14] S. MacDonald, D. Szafron, and J. Schaeffer. Object–oriented pattern–based parallel programming with automatically generated frameworks. In *Proceedings of the 5th USENIX Conference on Object–Oriented Technology and Systems*, pages 29–43, 1999.

[15] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating parallel program frameworks from parallel design patterns. In *Proceedings of the 6th International Euro–Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 95–104. Springer–Verlag, 2000.

[16] B. Massingill, T. Mattson, and B. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop*, 1999.

[17] F. Matthijs, W. Joosen, B. Robben, B. Vanhaute, and P. Verbaeten. Multi–level patterns. In *Object–Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357 of *Lecture Notes in Computer Science*, pages 112–115. Springer–Verlag, 1998.

[18] 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.

[19] 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.

[20] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern–Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley & Sons, 2000.

[21] M. Sefika, A. Sane, and R. Campbell. Monitoring compliance of a software system with its high–level design models. In *Proceedings of the 18th International Conference on Software Engineering*, pages 387–396, 1996.

[22] A. Singh, J. Schaeffer, and M. Green. A template–based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, 1991.

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

[24] 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*, pages 230–240, 1996.

[25] M. Snir, S. Otto, S. Hess-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[26] D. Szafron and J. Schaeffer. An experiment to measure the usability of parallel programming systems. *Concurrency: Practice and Experience*, 8(2):147–166, 1996.

[27] L. Tahvildari and A. Singh. Impact of using pattern–based systems on the qualities of parallel applications. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1713–1719, 2000.

[28] G. Wilson. Assessing the usability of parallel programming systems: The Cowichan problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, 1994.

[29] 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.

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