Pattern-based Parallel Programming

S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, K. Tan
Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2E8
Email: {bromling,stevem,janvik,jonathan,duane}@cs.ualberta.ca

February 8, 2002

Abstract

The advantages of pattern-based programming have been well-documented in the sequential literature. However patterns have yet to make their way into mainstream parallel computing, even though several research tools support them. There are two critical shortcomings of pattern (or template) based systems for parallel programming: lack of extensibility and performance. The patterns supported by a tool are typically limited in number or scope, thereby narrowing the applicability of a given system. As well, patterns usually offer generic solutions, which incur additional runtime overhead that impacts performance. This paper describes our approach for addressing these problems in the CO_2P_3S parallel programming system. CO_2P_3S supports multiple levels of abstraction, allowing the user to design an application with high-level patterns, but move to lower levels of abstraction for performance tuning. Patterns are implemented as parameterized templates, allowing the user the ability to customize the pattern to meet their needs. CO_2P_3S generates code that is specific to the pattern/parameter combination selected by the user. The MetaCO_2P_3S tool addresses extensibility by giving users the ability to design and add new pattern templates to CO_2P_3S. Since the pattern templates are stored in a system-independent format, they are suitable for storing in a repository to be shared throughout the user community. The CO_2P_3S/MetaCO_2P_3S combination is unique in the parallel computing world.

Keywords: parallel programming environment, design patterns, frameworks, meta-programming.
Technical Area: Programming Methodology and Tools

1 Introduction

Parallel programming is harder than sequential programming. Issues such as communication, synchronization, load balancing, deadlock, and granularity are additional complications that increase the effort required to develop correct, high-performance parallel applications. Despite two decades of parallel tools research, many of the ideas that have improved sequential programming productivity have yet to make their way into production parallel tools. Message passing libraries (e.g., MPI [23]) and compiler directives (e.g., OpenMP [8]) represent the state of the art.

There have been numerous attempts to develop high-level parallel programming tools that abstract away much of the parallel complexity. Several tools require the user to write sequential stubs, with the tool inserting all the parallel code [19, 22, 17, 4, 2]. Despite these (often large) efforts, high-level parallel programming

tools remain academic curiosities that are shunned by practitioners. There are two main reasons for this (others are explored in [20]):

- Performance. Generic tools generally produce abstract parallel code with disappointing parallel performance. Some developers may use a tool to generate the first draft of a (correct) implementation and then hand-tune the code by replacing many abstractions by application-specific code that runs faster. Unfortunately, most tools do not provide support for incremental code tuning (e.g., by making all the library code available in a usable form).

- Generality. The tools are usually suitable only for a small class of applications. If an application is not directly supported by the capabilities of the tool, then the developer cannot use the tool at all. Most tools do not support user-defined extensions to their capabilities.

For many developers, these are the main reasons why MPI and OpenMP are so popular: the user has complete control over the performance of their parallel application and the tools are general enough to be usable for a wide variety of applications.

This paper discusses the CO_2P_3S[1] parallel programming environment, designed specifically to address the performance and generality problems of current parallel programming tools [12, 13]. Three key features of CO_2P_3S contribute to solving these problems: parameterized frameworks, a model that supports multiple layers of abstraction, and a tool that supports the creation of new parallel design pattern templates.

A CO_2P_3S user expresses an application's concurrency by selecting one or more parallel design pattern templates. However, for each design pattern template, the user selects a series of parameter values that customizes the design pattern template for a specific application. These parameters are the first key feature of CO_2P_3S. The existence of these parameters allows CO_2P_3S to provide general enough design pattern templates to support a wide range of applications, while facilitating the generation of a non-generic efficient code framework. All application-specific code is entered as sequential stubs that are called by the framework.

Most high-level parallel programming tools use a programming model that suffers from a lack of "openness" [20]. The code generated by these tools is often difficult to tune for performance. In some cases, parts of the code are not available to the user. In other cases, all the code is available, but it is not very human-readable. Too much effort may be required to understand the underlying software architecture. The second key feature of CO_2P_3S is its open and layered programming model. All of the code is available to the user and it is organized into three layers to make the software architecture understandable at three different levels of abstraction. Performance tuning proceeds from the simplest most abstract top layer to the more detailed lower layers. When sufficient performance is obtained, no further code details need to be learned. This approach provides performance gains that are commensurate with the effort expended.

The third key to CO_2P_3S performance and generality is the MetaCO_2P_3S tool. For pattern-based programming, current academic tools only support a small number of patterns and, with only a few exceptions, do not allow the creation of new patterns. MetaCO_2P_3S is a graphical design pattern template editor. It supports generality by allowing a pattern designer to both edit existing CO_2P_3S pattern templates and to add completely new pattern templates. New pattern templates are first class in that they are just as general and powerful as the built-in CO_2P_3S patterns. In addition, MetaCO_2P_3S stores pattern templates in an XML format that is independent of CO_2P_3S so they can be used in other parallel programming systems

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

as well. Finally, the MetaCO$_2$P$_3$S tool also supports performance, by making it easy to design pattern templates with many parameters that can be used to generate efficient code.

This paper makes the following contributions to the evolution of high-level parallel programming environments:

1. CO$_2$P$_3$S, a parallel application development tool that provides a layered programming model with multiple user-accessible abstractions for writing and tuning parallel programs. The frameworks generated by CO$_2$P$_3$S are customized to the parameter combination selected by the user, eliminating unnecessary run-time overheads.

2. MetaCO$_2$P$_3$S, a parallel design pattern development tool. MetaCO$_2$P$_3$S supports the development of platform-independent pattern templates that can be imported into CO$_2$P$_3$S. The tool simplifies the pattern designer's task of specifying a pattern template, its parameters, and the code that is generated.

3. The first parallel programming system that supports the creation and usage of tool-independent parallel design patterns.

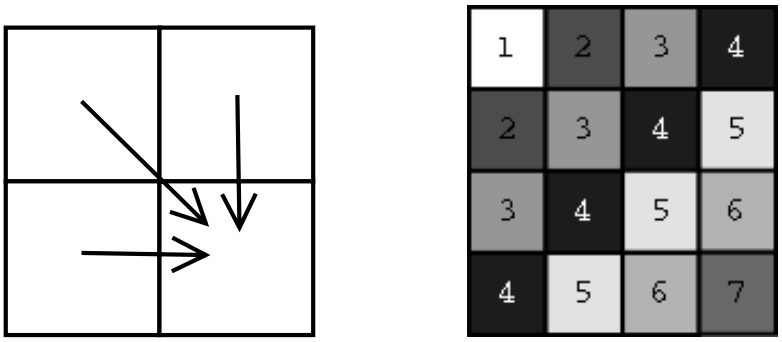
The CO$_2$P$_3$S and MetaCO$_2$P$_3$S combination is unique in the parallel computing world. Building a pattern-based parallel programming system is already a difficult task. Ensuring that the same environment can support the addition of new patterns adds further complications. Our community needs a meta-programming tool that enables the creation of new patterns and supports the modification of existing ones. However, we need more than this. If pattern-based programming systems are going to be accepted by the parallel programming community, the patterns must be transferable and re-usable from system to system. This means that the patterns must be expressible in a system-independent manner, and stored in a central repository. We need "open-patterns" in addition to open-source code. Only in this way can the high-level parallel tools research community create a critical mass of patterns that will interest practitioners. CO$_2$P$_3$S and MetaCO$_2$P$_3$S is a step towards realizing this vision.

Section 2 presents a sample application that is used throughout the paper. Section 3 describes CO$_2$P$_3$S and Section 4 discusses MetaCO$_2$P$_3$S. Section 5 compares CO$_2$P$_3$S and MetaCO$_2$P$_3$S to other related systems. Conclusions and future work are presented in Section 6.

2 Case Study: Genetic Sequence Alignment

Throughout this paper, we will use the sequence alignment problem from computational biology to illustrate how CO$_2$P$_3$S uses design pattern templates to parallelize applications. This problem is solved by finding an optimal alignment (with respect to a given scoring function) for a pair of DNA or protein sequences [7]. Typical algorithms for sequence alignment construct a dynamic programming matrix with the sequences on the top and left edges. A score is propagated from the top left corner to the bottom right. The value of each entry in the matrix depends on three previously computed values — north or above, west or to the left, and north-west or the above-left diagonal — as shown in Figure 1(a). Once the algorithm has calculated all of the values in the matrix, the maximal cost path (optimal sequence alignment) can be obtained by tracing backwards through the matrix.

This problem was given to a parallel programming expert, who was then asked to parallelize the application using CO$_2$P$_3$S. The expert quickly identified a Wavefront parallel design pattern. The Wavefront



(a) Score propagation.

(b) The wavefront computation ordering.

Figure 1: Solving the sequence alignment problem with a dynamic programming matrix.

pattern applies to problems where a computation needs to sweep breadth-first through a tree, with child nodes having data dependencies on their parents. The term *wavefront* describes the edge separating the processed nodes at the top of the tree from the nodes waiting to be processed. The dynamic programming problem is easily expressed as a wavefront due to the dependency of each matrix entry on three neighbors. Figure 1(b) shows how the data dependencies in Figure 1(a) can be transformed to a wavefront computation. Blocks with the same number can be computed concurrently after the blocks with smaller numbers that satisfy the data dependencies have been computed. CO_2P_3S has been used to satisfy many wavefront-like parallel problems including dynamic programming problems such as sequence alignment, computing a matrix product chain, and solving skyline matrices [1].

3 CO_2P_3S

The parallel computing literature is replete with familiar parallel structures. Concepts such as a pipeline, master-slave, work-queue, divide-and-conquer, and wavefront are structures that are well-known to experienced parallel programmers. Each concept immediately conjures up a vision of the concurrency that it represents. A parallel programmer who has used a master-slave structure to solve one application will use that experience to significantly reduce the effort needed to implement another master-slave application. In object-oriented computing, design constructs that can be re-used between applications are often expressed as design patterns [9], which capture design experience at an abstract level. By their nature, design patterns are applicable to different problem domains, each with its own individual characteristics and concerns. A design pattern is a description of a solution to a general design problem that must be adapted for each use. Once a user elects to use a design pattern, most of the basic structure of the application can be inferred.

CO_2P_3S uses object-oriented techniques to simplify parallel programming [12, 13, 14]. Developers begin by identifying the parallel design patterns that describe their application's basic structure. However, CO_2P_3S takes advantage of this knowledge in a more concrete manner. Specifically, parallel applications that use a similar pattern, not only share a design solution, they also have a considerable amount of common code, with only details that vary between applications. CO_2P_3S supports the re-use of this common code by

generating a framework of parallel structure code from the selected design pattern. Users then implement their application-specific code within this automatically generated framework that hides the entire parallel infrastructure. CO_2P_3S targets programmers looking for reasonable speedups in their sequential applications in return for a modest programming effort.

3.1 Frameworks

Frameworks are a set of classes that implement the basic structure of a specific kind of application [11]. This structural code defines the classes in the application and the flow of control through these classes. In general, a user of a framework must subclass the framework classes to provide implementations of key components and compose these classes into an application. In CO_2P_3S, the framework generates placeholders for sequential hook methods that are called by the framework. These methods are the only interface between the application-dependent code and the framework. The user need only implement the few hook methods to connect the framework to the sequential application code needed to solve the problem.

Frameworks are similar to libraries in some respects. However there is an important difference. When a library is used, a programmer must define the structure of the application and make calls to the library code. Conversely, a framework defines the structure of an application and the programmer supplies code for specific application-dependent routines. Frameworks allow programmer to reuse the overall design of an application, and capitalize on the use of object-oriented techniques such as encapsulation and code reuse. Frameworks can reduce the effort required to build applications [11].

We combine frameworks and design patterns by considering the application domain of the framework to be the implementation of a pattern. For instance, consider the description of a Wavefront design pattern. Most wavefront algorithms share the same basic structure with only a few application-dependent properties. We can encapsulate this structure into a framework that implements a basic wavefront. The structural code uses classes that define the application-dependent properties of a wavefront without implementing them, such as the function that computes the value of an element from the value of its predecessors. A user of this wavefront framework supplies these properties by either providing subclasses that contain the needed implementation or by composing framework-supplied classes that modify the basic structure to better fit the application.

3.2 Pattern Templates

It is non-trivial to generate a framework from a design pattern. A design pattern actually represents a family of solutions to a set of related problems. For example, the sequence alignment wavefront problem has specific data dependencies (left, above, and above-left) and the computation shape is an entire rectangular matrix. Other problems can be solved using a Wavefront pattern with different dependencies and different data shape, such as a band of a rectangular matrix or a tree of dependent nodes. It is unreasonable to expect a single design pattern to generate a single code framework that implements a solution to the entire family of problems represented by the design pattern. There appear to be only two options. One option is to specialize the generic design pattern into a series of design patterns that solve specific kinds of problems. However, this approach fails to recognize and take advantage of the common design and code that appears in the specialized patterns and frameworks. The other option is to require a single generic design pattern

to generate a single highly-abstracted code framework that can be specialized at run-time using a series of parameter values that describe the application. However, this approach yields code that is complex and inefficient. Recall that the framework code defines the flow of control, independent of the user's code and application. In the case of a wavefront, it is hard to imagine a single efficient flow of control that can be used to traverse either a complete matrix or a tree for any given application with wavefront-like parallelism.

The CO_2P_3S solution is to start with a design pattern template instead of a design pattern. A *design pattern template* is a design pattern that has been augmented by a series of parameters whose values span the family of design solutions represented by the design pattern. For example, in the CO_2P_3S Wavefront design pattern template, two of the parameter are the dependency set and the data shape. It is essential that the parameter values be set by the user before the framework code is generated. This allows the design pattern template to span a large set of application programs while enabling it to generate efficient framework code for each specific application.

CO_2P_3S generates correct framework code from a design pattern description of the application structure. The generated code is specific to the pattern/parameter combination specified by the user. In contrast, other systems provide the ability to specify the structure with patterns but rely on the user to write application code that matches the parallel specification. High-level parallel programming tools have failed to become mainstream for a number of reasons, including lack of generality and mediocre application performance. The concept of design pattern templates introduced in CO_2P_3S is a major step in addressing these concerns.

The implementation details of the framework are hidden from the programmer. To change the parallel structure of a program, the user changes a parameter for the pattern template and regenerates the framework code. Although this new framework may introduce additional hook methods that need to be implemented, hooks that were previously implemented are included in the regenerated framework automatically. Alternately, to obtain larger structural changes, the programmer may select a completely different design pattern. In this case, the programmer will need to implement different hook methods; application code will need to be moved from the old hook methods to the new ones.

CO_2P_3S does not provide a tool for users to deduce appropriate pattern templates for their application. Research into pattern languages [15] could eventually address the pattern selection problem.

3.3 A Layered Programming Model

One of the key aspects of CO_2P_3S is its separation of parallel code from user code. Communication and synchronization constructs are hidden from the user in the generated framework. This helps to prevent the user from writing an incorrect parallel program, and greatly simplifies their implementation effort. To ensure that parallelism hiding does not limit CO_2P_3S users, the programming model is "open", exposing all components of the generated program. CO_2P_3S supports three layers of abstraction, allowing the user to design at a high-level, and move down to lower-levels for performance tuning (if needed). At the highest layer, the user has no access to the parallel code. In the Intermediate Code Layer, a high-level, explicitly parallel programming language allows the user to manipulate the parallel structural code. At the lowest layer, native object-oriented code for the entire program is provided.

Patterns Layer This layer promotes the rapid development of structurally correct parallel programs. The user selects a parallel design pattern template from a palette of supported templates. This template

represents a family of frameworks for a given design pattern. The user can select the member of this family that is best suited for an application by specifying application–dependent template parameters. Once the parameters have been specified, the template is then used to generate object–oriented framework code implementing the pattern indicated by the template. This code consists of abstract classes that correctly implement the parallel structure of the pattern template together with concrete subclasses that are used to insert application–dependent sequential code (i.e., a structurally correct parallel program). Structural correctness is ensured by restricting access to the abstract structural classes; the user can only implement the hook methods in the concrete classes, which do not require any parallel code to work properly. A complete application 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 object-oriented language.[2] The abstract structural classes are implemented using this language and are made available to the user. The user can modify the generated structure, write new application code, or tune the structure.

Native Code Layer At this 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 can tailor the libraries for the application requirements or for the execution environment.

The user can use the Patterns Layer to quickly build a correct program, and then move down to the Intermediate and Native Code Layers to further tune the program, if necessary.

3.4 Wavefront Application

We now illustrate application development with CO_2P_3S using the sequence alignment application. A more detailed description of using CO_2P_3S can be found in [12].

Figure 2 shows the main CO_2P_3S window with the Wavefront pattern template selected [1]. Using the pattern window, a CO_2P_3S user can select parameters to customize a pattern template instance so that it matches their application's needs. The Wavefront supports three design parameters and three performance parameters. The design parameters are:

1. The name for the the Wavefront element class.

2. The shape of the element matrix. The default choice is a *full matrix* shape in which all elements of a rectangular matrix are computed. The second choice is *triangular*, which supports computations over half of a matrix. The third choice is *banded* which represents computations along a band of elements centered around the diagonal.

3. The dependency set for an element. Figure 1(a) illustrated the dependency set for the sequence alignment application. Not all groups of directions form legal dependency sets. For example any set that contains opposite directions is illegal since they generate cyclic dependencies that result in

[2]Not to be confused with the intermediate code used by compilers.

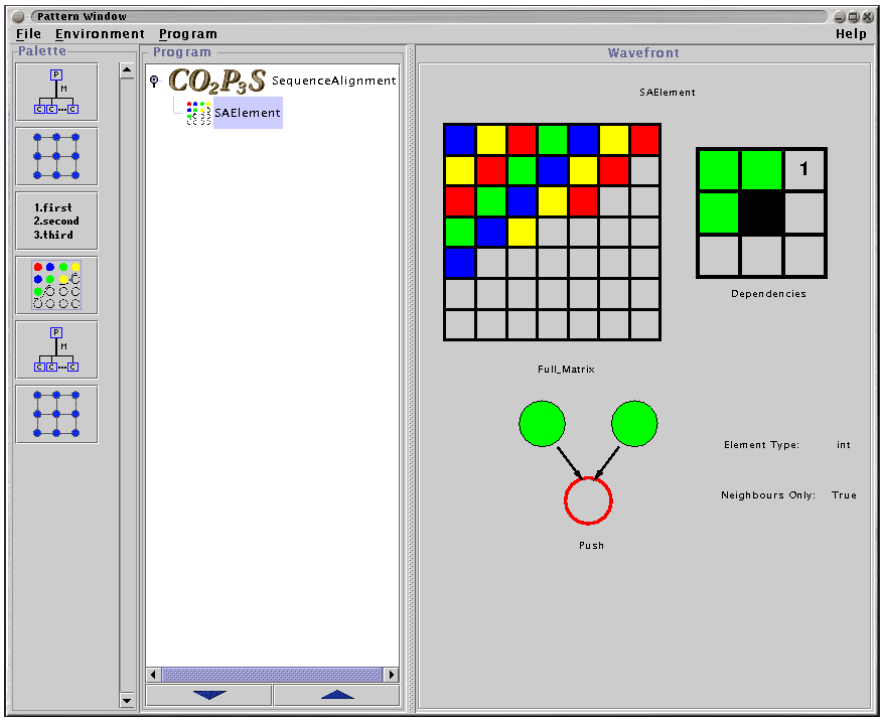


Figure 2: The Wavefront pattern template in CO_2P_3S.

deadlock. The user interface prevents a programmer from selecting illegal dependency sets. On the right-hand side of Figure 2, the dependencies graph for the sequence alignment problem is shown.

The performance parameters are:

1. The notification method used to inform elements that a dependency constraint has been satisfied. One choice is *push*, where elements signal their dependents upon completion. The other choice is *pull*, where dependents poll their prerequisites for completion status. Figure 2 shows that the push parameter has been selected.

2. The data type of wavefront elements. If the type is primitive, then the user selects the specific primitive type. Static methods are generated in the element class with the appropriate argument types and return types. If the element type is an object then instance methods are generated to operate on instances of the element class.

3. Whether the application needs access to non-neighboring elements. For example, in Figure 1(a), an element is directly dependent on three neighbors. If the element can actually be computed from the values of these three neighbors, the neighbors-only parameter would have the value true. However, if the element required all elements to the left of it, as well as the above and above-left neighbors, the neighbors-only parameter would be false. Notice that the dependency set would still be {above, left, above-left} since once these values are available, the element can be computed.

In Figure 2, the parameters have been set to the values used for the sequence alignment problem.

The design parameters show the flexibility of the approach, since a large family of different parallel structures can be built from one pattern template. The performance parameters indicate that the generated

code is not generic; each parameter setting allows the system to be more specific in the code that is generated, avoiding unnecessary run-time overheads.

After customizing the pattern template by selecting the appropriate set of parameters, the user requests that the framework code be generated. The current pattern templates in CO_2P_3S are configured to generate shared-memory Java code. At the heart of the wavefront implementation is a (hidden) driver that represents the control flow of the parallel code. Figure 3 shows an abstracted version of the wavefront driver. The only class the user implements is the *WavefrontElement* class. This class has three responsibilities: providing an initialization method that is called for each wavefront element (*initialize*), implementing the wavefront operations on a wavefront element (*operateCorner*, *operateTop*, *operateLeft*, *operateInterior*), and supplying a method to be used for reducing the matrix after the computation is complete (*reduce*). The user is supplied with stubs for the appropriate hook methods depending on the parameters specified at framework creation time. The framework invokes the appropriate method based on the location of the element. Each method has appropriate dependent elements as arguments and returns the computed element.

Figure 4 illustrates the user's view of the framework, showing a class that represents a single node in the wavefront. Users are not allowed to edit the class in this window. This prevents them from modifying method or class signatures. Instead, hyper-links are provided for user-modifiable locations in the code. The inset window is the result of following one such hyper-link. This allows the user to enter the selected method's body.

In Figure 3, note that the generated code skeleton has been specialized to take into account the computation dependencies (above, left, above-left) ensuring that the different cases are called under the right conditions and with the right parameters. This example is intended to emphasize the point that each pattern/parameter combination results in customized code. It also illustrates the need for different layers of abstraction. For example, the code in Figure 3 is non-optimal in the sense that there are far more interior nodes in the matrix than there are corner nodes. Reorganizing the *if* statements to reflect this will result in a (slightly) faster program. This change cannot be made in the Patterns Layer of CO_2P_3S; the user cannot touch the wavefront driver code. At the Native Code Layer, all the code is exposed to the user, who can then make whatever changes are appropriate. Although this is a trivial example, it does illustrate the need for multiple layers of abstraction in the programming model.

From the pattern designer's point of view, the wavefront can be implemented using a work queue, where nodes at the edge of the wavefront whose data dependencies have been satisfied are available. A user's view into a wavefront framework requires only that they provide the node processing implementation. They need not know about the wavefront driver and how it is implemented. The CO_2P_3S framework includes all the communication and synchronization needed by the application, and automatically divides the matrix into blocks to ensure reasonable granularity. All the user sees is their *initialize*, *operate*, and *reduce* routines for a single wavefront node.

The Wavefront pattern template in CO_2P_3S was used to implement the dynamic programming matrix algorithm for sequence alignment. Two sequences of 10,000 random proteins each were used as test data. The sequential and parallel implementations of the algorithm were run using a Java 1.3 VM on a four-processor shared-memory SGI O2. The *push* and *pull* notification parameter settings were both used independently as a comparison. Table 1 shows the execution times for 20 runs of each implementation. The parallel speedups are compared in Figure 5. Given a problem large enough to amortize the cost of having a lack of work to

```
public void execute(){
    this.initialize();
    while(!queue.isEmpty()){
        /* "work" is a WavefrontElement */
        work = queue.getWork();
        if(work is the corner) /* First element – no neighbors */
            work.operateCorner();
        elsif(work is on top edge) /* Top row – one neighbor */
            work.operateTop(location, leftValue);
        elsif(work is on left edge) /* Left column – one neighbor */
            work.operateLeft(location, aboveValue);
        else /* Interior – three neighbors */
            work.operateInterior(location, aboveValue, leftValue, diagonalValue);
    }
    this.reduce();
}
```

Figure 3: Wavefront code skeleton.

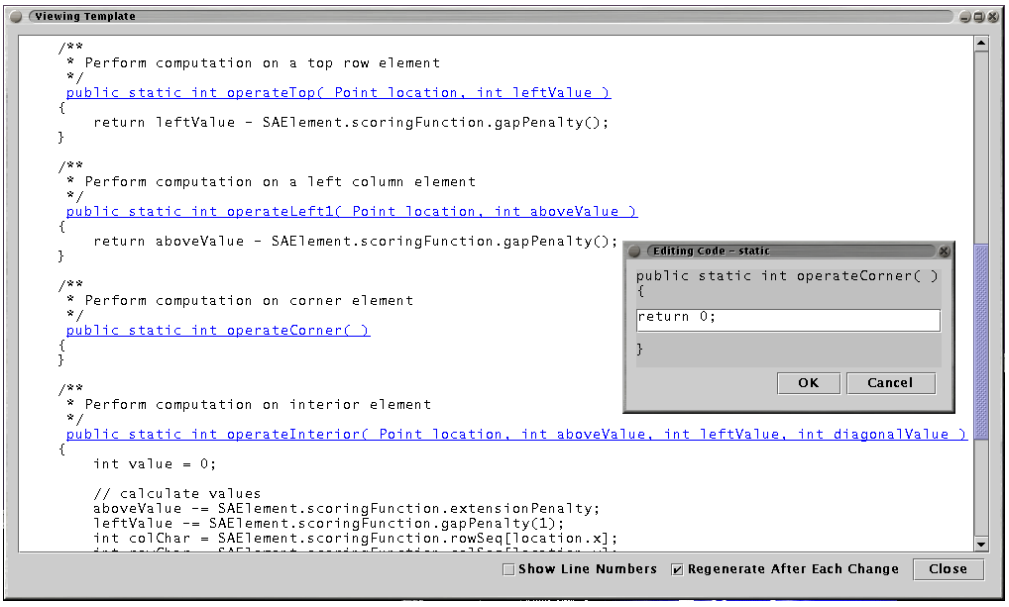


Figure 4: The user's view into the framework generated by CO_2P_3S.

Notification	Execution Time (seconds)			
	Seq	2P	3P	4P
Push	229.0	117.3	83.2	65.4
Pull	230.1	118.5	83.5	66.4

Table 1: Execution times using wavefront for sequence alignment.

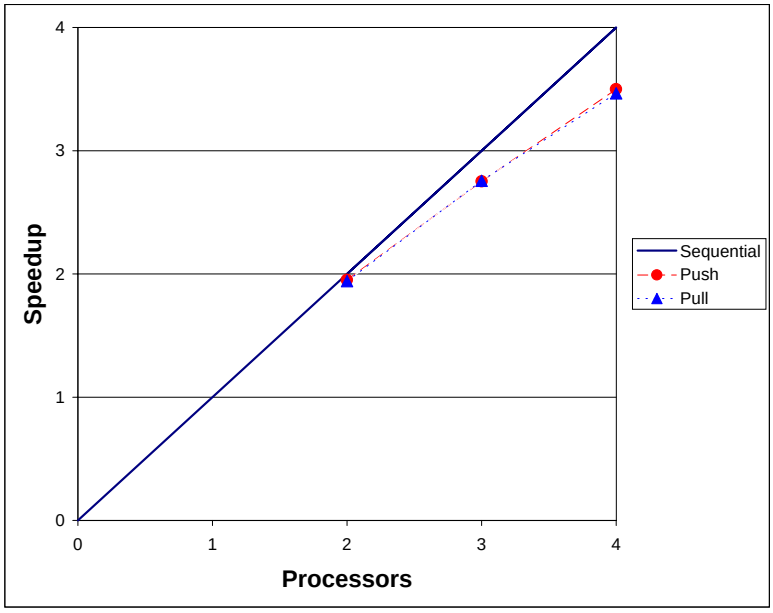


Figure 5: Speedups using wavefront for sequence alignment.

do at the beginning and end of a wavefront computation, one would expect to see close to linear speedups.

Given a functional sequential sequence alignment program, a user was able to get the parallel version running using CO_2P_3S in two hours. Of course, this time is not necessarily representative of the cost of doing any application in CO_2P_3S. Rather it shows that CO_2P_3S has a small learning curve, and that abstraction can have a major impact in the cost of program development. We have performed a user study that confirms these observations [13].

For this particular application, the push/pull choice did not make a significant difference in performance. Was this obvious in advance? The ability to change a fundamental property of the parallelism – pushing versus pulling data – without the user having to write any additional code is a powerful capability. The user can experiment with different parameter combinations to find the settings that offer the best performance. Once this is done, the user then can move to the Intermediate or Native Code Layers of CO_2P_3S to do further performance tuning (e.g., modifying the default blocking algorithm to improve the granularity).

To summarize, compared to previous pattern-based parallel programming efforts performance can be enhanced by:

1. A layered programming model, with the appropriate code exposed to the user at each level. This gives the user the ability to design, develop, test, and experiment at the highest level of abstraction, and move to lower layers of abstraction (if needed) to make incremental improvements.

2. Generalized patterns—pattern templates—that can be customized by parameters. The code generated by a parameter setting is tailored to that particular setting, avoiding the overhead of generic code.

4 MetaCO$_2$P$_3$S

The previous section illustrated program development using the CO$_2$P$_3$S Wavefront pattern template. However, the original version of CO$_2$P$_3$S did not include a Wavefront pattern template. If tools such as CO$_2$P$_3$S do not support the pattern required by an application, then the application developer would be forced to use a different tool. The likelihood of the user coming back to give a CO$_2$P$_3$S-like tool a second chance would be small.

Before the sequence alignment program could be implemented in CO$_2$P$_3$S a new parallel design pattern template had to be added. MetaCO$_2$P$_3$S is a tool that supports adding new pattern templates to CO$_2$P$_3$S. Any design-pattern-based tool will be limited by the set of patterns that it supports. With most of the parallel pattern-based tools in the literature, if a user's application does not match the suite of patterns available in the tool, then the tool is effectively useless.

The previous comments motivate the need for generalizing our pattern-based parallel programming system by providing extensibility. The applicability of any template-based environment is limited by the scope of the templates provided. If an application cannot be implemented using a given programming environment, it calls into question the utility of that tool. Programmers are unlikely to invest effort learning an environment that may not meet their needs on future projects.

To address the extensibility problem, we have created a tool that allows parallel and object-oriented programming experts, called *pattern designers*, to create new pattern templates. The new pattern templates are *first-class*, meaning they are indistinguishable in form and equivalent in function to the pattern templates included with CO$_2$P$_3$S. The tool simplifies the task of building pattern templates. In particular, it facilitates the parameterization and code generation for the templates.

It is the responsibility of the pattern designer to identify new design patterns. This involves isolating a recurring pattern and the various forms that it can take based on pattern parameters, then creating a framework that hides the parallelism details. The designer should take note of the aspects of the framework that are affected by different parameter settings. At this point, the pattern is ready to be formalized as a pattern template using MetaCO$_2$P$_3$S. Analogous to the manner in which CO$_2$P$_3$S makes it easier to write parallel programs using pattern templates, MetaCO$_2$P$_3$S makes it easier to write pattern templates for CO$_2$P$_3$S.

The following sections provide a summary of the pattern template creation process for CO$_2$P$_3$S. Figure 6 illustrates the MetaCO$_2$P$_3$S design process and how this information is used in CO$_2$P$_3$S. A more thorough explanation of the process is available in [5].

4.1 Pattern Template Creation with MetaCO$_2$P$_3$S

MetaCO$_2$P$_3$S is launched from the CO$_2$P$_3$S GUI and guides the pattern designer through the pattern template creation process. The pattern designer is required to supply three kinds of information to MetaCO$_2$P$_3$S:

Class names: For each class in the pattern template's framework, a placeholder class name must be sup-

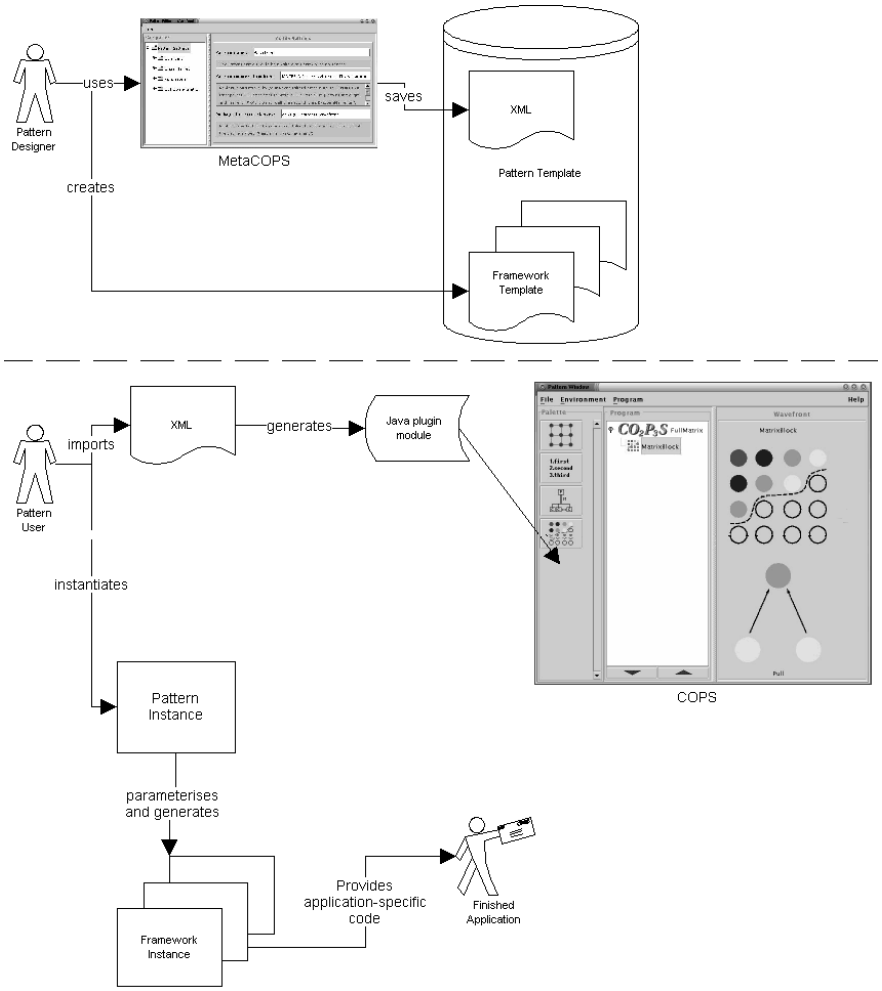


Figure 6: Overview of CO_2P_3S and MetaCO_2P_3S.

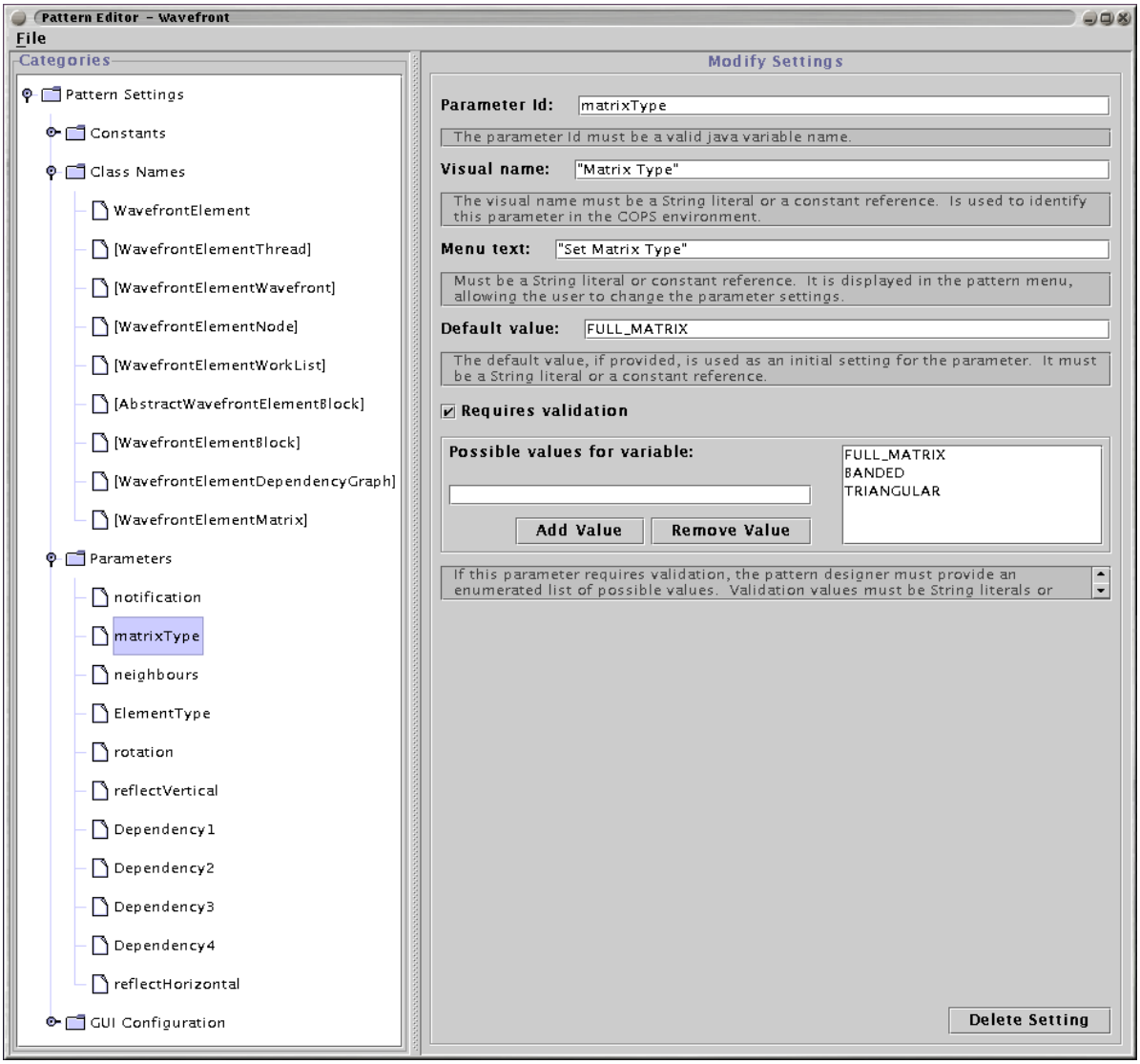


Figure 7: Specifying a pattern template in MetaCO_2P_3S.

plied. Since multiple instances of a single pattern template can be used in a CO_2P_3S application, the placeholder names are replaced by unique run-time names in framework instances. This is achieved by requiring that CO_2P_3S users provide at least one user class name for their pattern template instances. The remainder of the framework classes are uniquely named by adding suffixes or prefixes to the user-supplied class name.

Parameters: There are three types of parameters that can affect the framework implementation of a pattern template:

1. *Basic Parameters* are the most common. They consist of an enumerated list of choices supplied by the pattern designer. This allows boolean-like switches, as well as more elaborate list choices. For example, in the Wavefront pattern template, *notification* is a basic parameter with two values, push and pull. In our Mesh pattern template, the user can specify the *number of neighbors*, which can be set to either four-point or eight-point.

2. *Extended Parameters* deal with the less common case where the parameter value is of an arbitrary form. Pattern designers must provide extra information for each extended parameter, including a way for CO_2P_3S users to supply the run-time parameter settings, and the manner in which a given parameter setting affects framework code generation. Code generation is discussed further in Section 4.3. The Distributor pattern template available in CO_2P_3S, which parallelizes the computation of methods with array parameters, has a list parameter composed of extended parameters. Each entry in the parameter list is a method signature with optional distributions {pass-through, block, neighbor and striped} for method arguments that are arrays.

3. *Extended List Parameters* handle a common situation wherein the CO_2P_3S user supplies a list of values, which may in turn be basic or extended parameters. One common example is a list of method signatures provided by the CO_2P_3S user that need to be added to the generated framework code. An example is the Wavefront pattern template's dependency list.

   Figure 7 shows an example of the specification of the classes and parameters for our application. All class names which are framework classes (and not accessible to the user) are shown with braces around the name. Template classes, which the user is required to provide the code for, do not have the braces. In the Wavefront pattern, there is only a single template class: *WavefrontElement*. Each pattern template parameter is specified with attributes including its legal values. In this example all parameters are basic parameters and the values are previously specified constants.

GUI configuration: Figure 2 shows a graphical representation of a CO_2P_3S pattern template. It is recommended that pattern designers follow this model, and display images or textual data to indicate the parameter settings. The GUI configuration section of MetaCO_2P_3S, illustrated in Figure 8, simplifies this task. GUI elements are either labels (shown as files) or images (shown as folders), and they may be dependent on the value of a pattern parameter. For example, the matrixType label changes based on the value of the matrixType parameter. Similarity, the image displayed may be dependent on the value of a parameter requiring different images to be specified for each parameter value, hence the folder representation in Figure 8.

4.2 Framework Generation

Given a pattern template and a specific parameterization, CO_2P_3S generates an appropriate object-oriented framework instance. The pattern designer must create a framework template to define the code to be generated for the pattern template. It is the pattern designer's responsibility to ensure that the framework

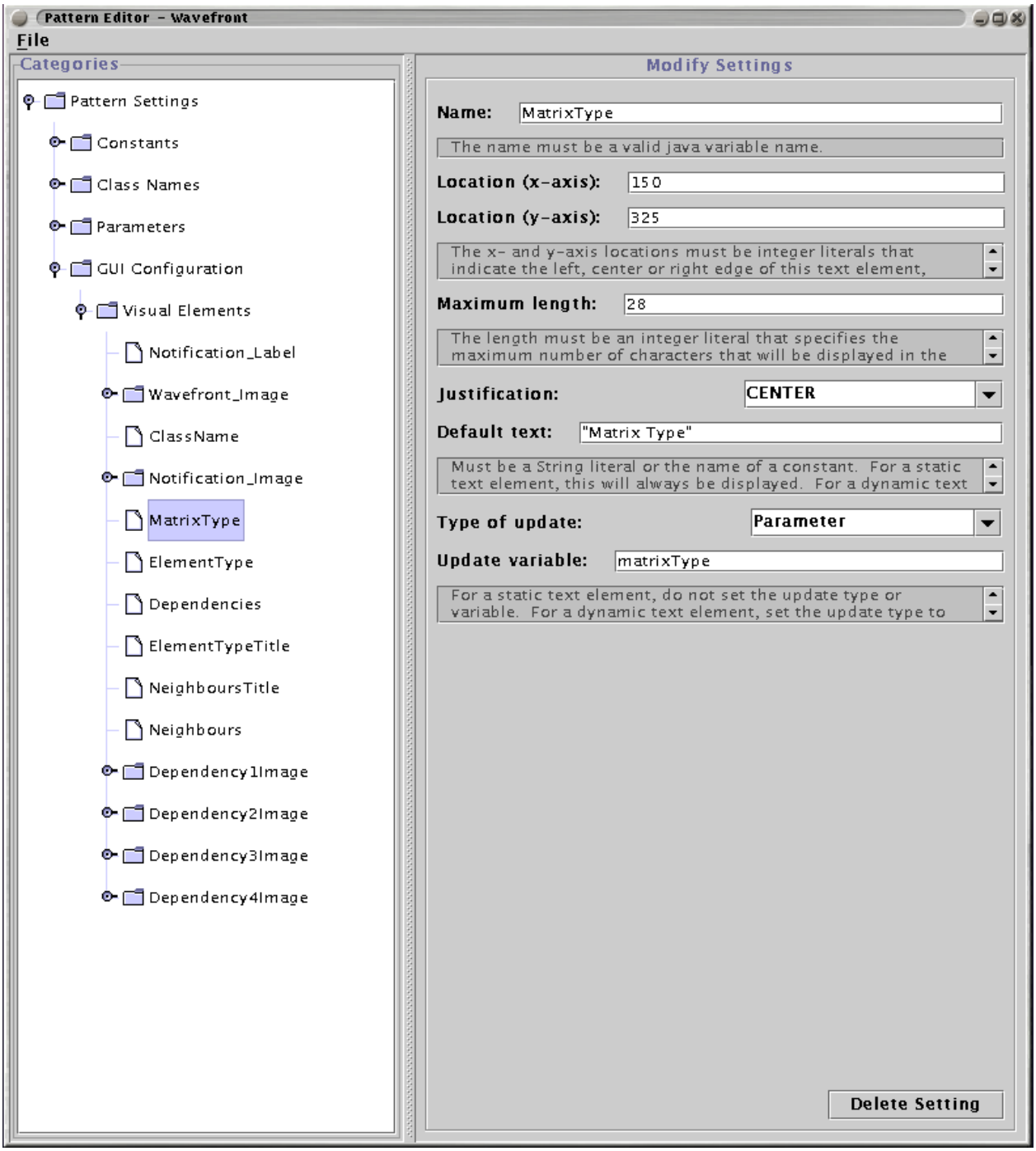


Figure 8: GUI configuration using MetaCO_2P_3S.

template is set up correctly for code generation, and that the generated frameworks are error-free. An annotated source code template must be written for each of the classes in the framework. The annotations supply the information needed to make the following transformations during framework generation:

- Placeholder class names in the annotated source files must be replaced with the unique names that are supplied by the CO_2P_3S user.
- Methods or variables may be selectively generated based on the user's basic parameter settings. The pattern designer must specify the combination of parameter settings that allow a given construct to be generated.
- Portions of method bodies may be selectively generated based on the basic parameter settings.
- New methods or sections of method bodies may be generated based on extended or list parameter settings.
- The pattern designer must select at least one framework class that the CO_2P_3S user can use to add application-specific code. Annotations must be supplied to allow CO_2P_3S to generate a non-modifiable version of the class, with hyper-links in the user modifiable locations.

Further details on the annotation and code generation implementations are in Section 4.3.2.

4.3 Architecture for Pattern Template Design

This section discusses the architecture of each of the components of the pattern template creation process.

4.3.1 XML for Pattern Descriptions

The pattern descriptions created by MetaCO_2P_3S are in a system-independent XML (Extensible Markup Language) format. XML is a text-based mark-up language descended from SGML. It is similar to HTML but more general. Pattern designers can also store partially completed pattern descriptions in this format for later completion. A DTD (Document Type Definition) file comes with CO_2P_3S that describes the format of the XML pattern description file.

When a completed pattern description is imported into CO_2P_3S, the XML is converted into a compiled plug-in module. XSL (Extensible Stylesheet Language) is used to transform the pattern description into a Java source file, which is subsequently compiled and loaded into CO_2P_3S. Since the XML file is system-independent, it can be used in template-based environments other than CO_2P_3S.

4.3.2 Javadoc for Framework Generation

Javadoc is a tool included with the Java distribution whose original purpose was to generate API documentation for Java libraries in HTML [10]. Javadoc runs a modified Java compiler on Java source code files to parse the declarations and specially formatted comments. Javadoc comments have the following format:

```
/**
 * A description of the following Java construct.
 *
```

```
     * @sampleTag  a tag that is parsed by Javadoc
     */
    public void sampleJavaDeclaration()
```

Javadoc was eventually extended to allow pluggable *doclets*. Doclets are Java programs that satisfy a contract allowing them to receive the parsed data from a given Javadoc execution. This data includes the declarations from each of the parsed classes. Method bodies and field initializations are not provided, since they are ignored during Javadoc's parse.

CO_2P_3S framework generation is a source code to source code transformation. There are two inputs to the process. One is a set of Java source code files that have been annotated by the pattern designer for use by Javadoc. The other is the pattern template parameters selected by a CO_2P_3S user. We have created special tags for the pattern designer, to allow for each of the transformations mentioned in Section 4.2. Default method bodies are provided in separate files, since they are not parsed by Javadoc. The output is a framework instance that has been specialized to match pattern template parameter settings.

4.4 Wavefront Revisited

Since no Wavefront pattern template was available in CO_2P_3S for the sequence alignment problem, we had an opportunity to use MetaCO_2P_3S to create a new pattern template. Our parallel programming expert had no involvement in the research on MetaCO_2P_3S, which also made this exercise a test-bed for the usability of our tool. The following section provides a detailed description of the steps taken by our expert to create the Wavefront pattern template.

The first step was to specify the pattern description using MetaCO_2P_3S. After launching the tool, the pattern designer named the new Wavefront pattern template and supplied an icon to identify the pattern in CO_2P_3S. Next, the placeholder framework class names were supplied. One of these, called *WavefrontElement*, was selected as a class to be named by the user. The CO_2P_3S GUI requires at least one user-named class prior to framework generation. The *WavefrontElement* class was chosen as the public class to be edited by the user in the browser shown in Figure 4. Eight other framework classes were defined, the names of which were made dependent on *WavefrontElement* for their uniqueness, with prefixes and/or suffixes added to indicate their role in the framework. In CO_2P_3S, the placeholder class names defined in the pattern description are used to supply run-time names to the code generator.

All the parameters that impact the generated code were specified through the MetaCO_2P_3S GUI. This was an iterative process. Initially, a parameter set was chosen that met the needs of a specific application (sequence alignment in this case). Later on, as new applications arose that required wavefront parallelism, the pattern was generalized by modifying existing parameters and adding new parameters. MetaCO_2P_3S provides an interface that facilitates this process, and reduces the likelihood of the pattern designer introducing an error. The result is a Wavefront pattern template that has been used in three different applications [1].

The last step in the pattern description process was providing a GUI configuration. The result of this configuration is shown in Figure 2. At the top, a textual element is displayed that automatically updates to display the user name for the *WavefrontElement* class. In MetaCO_2P_3S, the pattern designer needed to provide the location for the text, and the framework class name to display. Below the class name is an image of a wavefront. The pattern designer provided the location and image name. The image below the

wavefront and the text element below that are both dynamic representations of the value of the *notification* parameter. Using MetaCO_2P_3S, the pattern designer defined the images to be used for *push* and *pull* to match the possible values of the parameter.

After providing the pattern description in MetaCO_2P_3S, the pattern designer needed to provide annotated framework source code for each of the defined classes. This was similar to writing normal Java source code, with a few exceptions, as follows:

- Some methods were tagged using Javadoc so that they could be conditionally generated based on the value of the parameter.
- In the *WavefrontElement* class, some methods were tagged to allow them to be edited by the CO_2P_3S user.
- Since Javadoc does not parse method bodies, the pattern designer saved them to separate files, each named after their method signature. Portions of some methods were set apart to generate only for a particular parameter setting.

At this point, the pattern template was completely specified. The pattern designer imported it into the CO_2P_3S environment, and tested the pattern template prior to implementing the sequence alignment program. The pattern template creation took only a few hours.

4.5 Validating MetaCO_2P_3S

To test the coverage and correctness of MetaCO_2P_3S, we tried to regenerate each of the pattern templates from the original CO_2P_3S. Every pattern template has been successfully regenerated, and the new standard CO_2P_3S distribution is the one generated by MetaCO_2P_3S.

4.5.1 Pattern Template Repository

For CO_2P_3S (or any other pattern-based tool) to be accepted as a viable environment for writing parallel programs, its pattern templates must cover a wide variety of parallel problems. If pattern-based programming is to have the impact in the parallel world that it is having in the sequential world, then our community needs to have "open-patterns" (analogous to open-source code). Only in this way can the high-level parallel tools research community create a critical mass of patterns that will interest practitioners. Since MetaCO_2P_3S enables the creation of new pattern templates, the coverage can be made arbitrarily wide. To facilitate the sharing of pattern templates, we propose that a repository be created. Since the pattern templates consist only of XML and Java files, they are system-independent, and can easily be packaged in a downloadable format for distribution on the Internet.

Another advantage to a central repository is the ability it provides for pattern templates to be refined as new problems identify undiscovered parameters or implementation possibilities. Our Wavefront pattern template went through several such iterations as we applied it to additional applications. For example, when implementing the matrix product chain application, we discovered the need to access non-neighboring elements in the matrix [1]. Our original pattern template did not support this. The modification was accomplished by adding a new parameter to the pattern template, and specifying the effect that this new parameter had on the generated framework code. The Wavefront pattern template has also been copied and

modified in the repository by a researcher to generate framework code that runs on a network of workstations rather than a shared-memory machine.

5 Related Work

Although many research groups have studied parallel template-based programming environments, few have addressed the need for extensibility. Most of these environments are not suited for extension. One exception is the DPnDP distributed message-passing programming environment [21]. Like CO_2P_3S, DPnDP design patterns are modular, and support a pluggable library. However, DPnDP does not provide a tool for creating new patterns, but rather specifies a C++ framework under which patterns can be built. Patterns created using DPnDP can have only a structural specification; all behavioral aspects, such as communication and synchronization, must be supplied by the DPnDP user. However, the patterns supplied with DPnDP automatically implement any pattern-specific behaviors. Therefore, new patterns may not have the same level of functionality and abstraction as those provided with DPnDP.

Another environment that provides template extensibility is Tracs [3]. It provides a tool that allows pattern designers to define architectural models using a formal graph to specify task and communication structures. The architectural model does not include a framework implementation.

Automatic code generation has been studied by many groups with different agendas. Given our parameterization needs and language choice, we took our inspiration from [16, 18]. Also, we studied the COGENT code generator [6], which uses macro expansion to generate framework code for design patterns from the Gang of Four [9].

6 Conclusions

In this paper, two obstacles to the widespread acceptance of high-level parallel programming tools are addressed. Although this represents an important step forward towards moving these tools from academia into practice, much work remains to be done.

Performance is an issue with every parallel tool, and this has been particularly acute with high-level parallel programming models. CO_2P_3S resolves many of the performance issues through the use of parameterized parallel design pattern templates, a layered model of abstraction, and a tool to manage the complexities of creating/modifying patterns. The performance will, in part, be constrained by the quality of the code generated by the pattern template. If the ideas presented in this paper catch on, then the community can work towards highly optimizing this process to ensure the best possible performance.

This paper discusses how extensibility, critically lacking in all template-based systems, can be addressed. We have created both a parallel pattern template builder and a structure for framework templates. Our tool generates pattern templates that are first-class, and integrate seamlessly with CO_2P_3S. Through system-independence we have enabled the creation of a pattern template repository. The best chance for making pattern-based parallel programming a reality in practice rests upon whether the research community can work towards realizing the pattern repository vision, regardless of whether CO_2P_3S or MetaCO_2P_3S is used.

Acknowledgments

This research was supported by the Natural Science and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).

References

[1] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating parallel programs from the wavefront design pattern. In *7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02)*, 2002. To appear.

[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] A. Bartoli, P. Corsini, G. Dini, and C. Prete. Graphical Design of Distributed Applications Through Reusable Components. *IEEE Parallel & Distributed Technology*, 3(1):37–51, 1995.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore. HeNCE: A Heterogeneous Network Computing Environment. Technical Report UT-CS-93-205, University of Tenessee, 1993.

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

[6] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[7] K. Charter, J. Schaeffer, and D. Szafron. Sequence Alignment using FastLSA. In *Proc. of the 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS'2000)*, pages 239–245, 2000.

[8] L. Dagum and R. Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

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

[10] Javadoc Tool Home Page. http://java.sun.com/j2se/javadoc/.

[11] R. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[12] S. MacDonald. *From Patterns to Frameworks to Parallel Programs*. PhD thesis, Department of Computing Science, University of Alberta, November 2001. Available at www.cs.ualberta.ca/∼systems.

[13] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. From Patterns to Frameworks to Parallel Programs. Submitted to Journal of Parallel and Distributed Computing. Available at www.cs.ualberta.ca/∼systems, December 2000.

[14] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating Parallel Program Frameworks from Parallel Design Patterns. In *Euro-Par 2000, Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 95–104. Springer-Verlag, August 2000.

[15] B. Massingill, T. Mattson, and B. Sanders. A Pattern Language for Parallel Application Programs. In *European Conference on Parallel Processing*, pages 678–681, 2000.

[16] 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.

[17] P. Newton and J. Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 167–177, 1992.

[18] M. Pollack. Code Generation using Javadoc. http://www.javaworld. com/javaworld/jw-08-2000/jw-0818-javadoc_p.html, August 2000.

[19] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel & Distributed Technology*, 1(3):85–96, 1993.

[20] A. Singh, J. Schaeffer, and D. Szafron. Experience with Parallel Programming Using Code Templates. *Concurrency: Practice & Experience*, 10(2):91–120, 1998.

[21] S. Siu. Openness and Extensibility in Design–Pattern–Based Programming Systems. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, August 1996.

[22] 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.

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