Pattern–based Object–Oriented Parallel Programming

Steve MacDonald, Jonathan Schaeffer, and Duane Szafron

University of Alberta, Edmonton, Alberta CANADA T6G 2H1
Email: {stevem,jonathan,duane}@cs.ualberta.ca

1 Introduction and Motivation

Over the past five years there have been several attempts to produce *template–based* (or, as they are now called, *pattern–based* [4]) parallel programming systems (PPS). By observing the progression of these systems, we can clearly see the evolution of pattern–based computing technology. Our first attempt, FrameWorks [9], allowed users to graphically specify the parallel structure of a procedural program in much the same way they would solve a puzzle, by piecing together different components. The programming model of FrameWorks was low–level and placed the burden of correctness on the user. This research led to Enterprise [7], a PPS that provided a limited number of templates that could be composed in a structured way. The programming model in Enterprise was at a much higher level, with many of the low–level details handled by a combination of compiler and run–time technology.

In this progression, we can see more emphasis placed on the *usability* of the tools rather than raw performance gains, as shown by Szafron and Schaeffer [12]. Each successive system reduced the probability of introducing programmer errors. However, since performance considerations cannot be ignored, each successive systems also supported incremental application tuning. A related performance issue identified by Singh *et al.* [10] is *openness*, where a user is able to access low–level features in the PPS and use them as necessary. This work led to a critical evaluation of pattern–based systems that provides the motivation for our new system, CO_2P_3S (Correct Object–Oriented Pattern–based Parallel Programming System, pronounced "cops").

In this paper, we present an architecture and model for CO_2P_3S in which we address some of the shortcomings of FrameWorks and Enterprise. Our continuing goal is to produce usable parallel programming tools. The first shortcoming we address is the loose relationship between the user's code and the graphical specification of the program structure. Enterprise improved on FrameWorks by verifying a correspondence between the parallel structure and the code at compile–time. However, we feel that forcing the user to write a program that conforms to an existing diagram is redundant. If the structure of the application is already known, then the basic framework can be generated automatically. This reduces the amount of effort required to write programs, while simultaneously reducing programmer errors even further. The CO_2P_3S architecture also supports improved incremental tuning. The architecture is novel in that it provides several user–accessible layers of abstraction. At any given time during performance tuning, a programmer can work at the appropriate level of abstraction, based on what is being tuned. This can range from modifying the basic parallel pattern at the highest level, to modifying synchronization techniques at the middle layer, to modifying which communication primitives are used at the lowest level. Our goal is an open system where the performance of an application is directly commensurate with programmer effort.

2 The Architecture of CO_2P_3S

The architecture of CO_2P_3S consists of three layers: `Patterns`, `Intermediate Code`, and `Native Code`. These layers represent different levels of abstraction, where the abstraction of each layer is implemented by the one underneath.

Each layer is transformed during compilation to the layer underneath. At the pattern layer, the developer selects a pattern using a graphical tool. The PPS then generates a template for the parallel application and the user is restricted to providing sequential application–specific code at particular locations in the generated template. At the pattern level, the PPS guarantees the correctness of the generated program by using conservative synchronization mechanisms that may not yield peak parallel performance.

Unlike Enterprise, CO_2P_3S provides programmer access to the second layer where the templates can be edited. From this layer down, the programmer is responsible for the correctness of the resulting program. To simplify this task, we provide a high–level parallel programming language that is an extension of existing OO languages (Java or C++). This superset includes keywords to denote parallel classes, specify concurrent activities and express necessary synchronization (using constructs such as asynchronous methods, threads, and futures).

Finally, the third layer is the native programming language augmented with a library that provides the services required by the first two layers. Users are given full access to all language features and library code.

We believe that providing intermediate levels of abstraction can provide several benefits. First, by generating correct template code at the first level, we can ensure that a user has a working parallel program before the tuning process begins. Second, it eases the tuning process by introducing the run–time system in smaller increments. These smaller increments provide better opportunities for novice or intermediate users to find a comfortable level of abstraction while still providing full access to the run–time system for experienced users. Lastly, it should always be possible to improve the performance of a program by using the abstraction of a lower level. If so, then the performance of an application should be more directly commensurate with programmer effort.

3 The Model

In this section, we more fully specify the model and demonstrate it using an example program. Our example shows some details of a generic mesh computation.

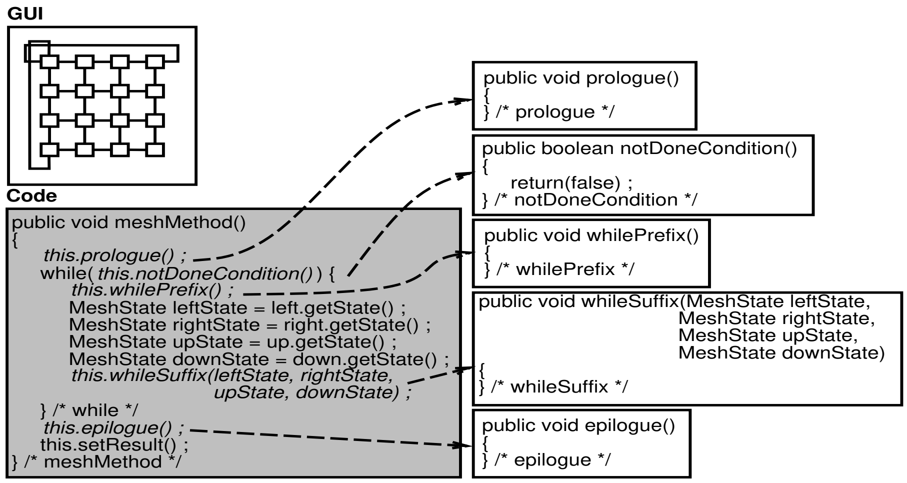

Fig. 1. The first layer of CO_2P_3S. Shaded code is generated by the system and cannot be modified. Italicized methods are null and must be implemented by the user. Skeleton implementations are shown on the right.

The first layer specification is shown in Figure 1. This layer consists of both a graphical representation of the pattern and the accompanying generated code. The pattern itself has several parameters that must be given. The parameters include the width and height of the mesh and its boundary conditions. The boundary conditions are the most interesting of these parameters since they can affect the user code. The reason for this can be seen in the accompanying code fragment. It shows the main loop in the evaluation of the mesh, with the shaded part representing the generated code and italicized code representing hook methods that the user must implement. In the hooks, the code uses all four mesh neighbours. However, if the mesh is not fully toroidal, then not all the neighbours will exist and the supplied code is not applicable. To correct this, we allow the user to specify different code for the different special cases. In this paper, for simplicity, we will assume the mesh is fully toroidal.

One critical feature of the first layer is that the generated code encapsulates the communication and only allows the user to operate on the data. This feature allows our PPS to make certain correctness guarantees, such as ensuring that each element of the mesh is referenced. The code in Figure 1 demonstrates this property by getting the state from the mesh neighbours and communicating the results of the computation to a *collector* object responsible for collecting the results (encapsulated within `setResult()`). Another critical aspect of this code is that we provide sufficient hooks for users to completely implement their programs. This requirement must be evaluated on a pattern–by–pattern basis. For the portion of the mesh in Figure 1, we provide five hooks. `prologue()` and `epilogue()` are executed before and after the mesh computation and can be used for initialization and cleanup, such as implementing instrumentation.

`whilePrefix()` and `whileSuffix()` are executed for every iteration of the mesh and can be used to preprocess local data and perform the desired mesh operation. Finally, `notDoneCondition()` specifies the terminating condition for an element of the mesh. The mesh stops executing when all the elements are finished. The default implementation of these methods is given on the right side of Figure 1.

From the first layer code, we generate the intermediate code of the second layer shown in Figure 2. The figure also contains an example of a method operating on data in the mesh, which would have been added by the user at the first level. For the mesh, the keywords `left`, `right`, `up`, `down` are defined to refer to the neighbours. The keyword `barrier` is also required to define the necessary synchronization for the mesh and is automatically inserted in the transformation of the mesh code from the first to the second layer. Other, more general keywords are also supported by CO_2P_3S.

```
public void meshMethod()
{
    this.prologue() ;
    while (this.notDoneCondition()) {
        this.whilePrefix() ;
        MeshState leftState = left.getState() ;  MeshState rightState = right.getState() ;
        MeshState upState = up.getState() ;  MeshState downState = down.getState() ;
        barrier ;
        this.whileSuffix(leftState, rightState,  upState, downState) ;
    } /* while */
    this.epilogue() ;
    this.setResult() ;
} /* meshMethod */

public void whileSuffix(MeshState leftState, MeshState rightState,
                        MeshState upState, MeshState downState)
{
    // Take the average ol four neighbours and gather timing information.
    this.value = (this.value + leftState.getValue() + rightState.getValue() +
                 upState.getValue() + downState.getValue()) / 5.0 ;
    this.stopTimer() ;
    this.gatherStatistics() ;
} /* whileSuffix */
```

Fig. 2. The second layer of CO_2P_3S. The italicized code are keywords inserted by the transformation between layers.

A feature of the generated code is that it is fully functional even if the user does not implement any of the hooks. In the mesh example, the default implementation for `notDoneCondition()` returns false so the mesh immediately exits. The main body of the mesh, if invoked, will retrieve the state from each of its neighbours and perform a null operation. By producing code that is capable of executing, we can verify that our patterns are correct and that the user will start with a parallel program that does not contain any communication errors.

The user can also edit the generated code at this level, which allows the structure of the mesh to be extended or modified. For example, we can add new neighbours not defined in the original mesh pattern. We can use this feature to define an arbitrary stencil to be convolved over the mesh.

Finally, from the second layer code of Figure 2, we generate the native code given in Figure 3. Native code replaces the keywords inserted in the second layer with library calls implemented by our run–time library. This replacement may simply replace the keyword with an accessor, as with the accessing the neighbours of the mesh. Others may be more complex, such as replacing `barrier` with a call to the thread group for the threads executing the mesh. Users can also use any available library call or language feature.

```
public void meshMethod()
{
    this.prologue() ;
    while (this.notDoneCondition()) {
        this.whilePrefix() ;
        MeshState leftState = this.getLeft().getState() ; MeshState rightState = this.getRight().getState() ;
        MeshState upState = this.getUp().getState() ; MeshState downState = this.getDown().getState() ;
        this.getMeshThreadGroup().barrier() ;
        this.whileSuffix(leftState, rightState, upState, downState) ;
    } /* while */
    this.epilogue() ;
    this.setResult() ;
} /* meshMethod */
```

Fig. 3. The third layer of CO_2P_3S. The italicized code represents replaced keywords. The code for other methods is accessible but not shown.

4 Evaluating the Model

In this section, we examine the 13 desirable characteristics of pattern–based PPSs described by Singh *et al.* and apply them to CO_2P_3S. Like Singh *et al.*, we break the characteristics into three categories. We also use their short names which are shown in bold in this text.

Structuring the Parallelism This category examines how users can structure the parallelism in their programs. There should be as few restrictions as possible.

CO_2P_3S addresses **separation** by expressing the parallelism diagrammatically and allowing application–specific code to be inserted into the pattern. In our case, the diagram is some form of *collaboration diagram*. Further, the CO_2P_3S approach of generating code that invokes user hooks provides more opportunity to re–use the hooks when the user changes the pattern. Since the hook code does not affect the communication flow of the program (because the communication code is generated and cannot be edited by the user), there is a greater possibility that this code can be used in another pattern.

We can allow the parallelism in a program to be **hierarchically** specified by allowing patterns to be substituted for the sequential components in a program. This composition can be compared to the Composite design pattern [4]. It provides a structured way of building complex program elements.

We can only address **independence** and **utility** once the set of templates has been decided upon. Typically, independence has been addressed by rigourously defining the inputs and outputs of each design pattern or by creating separate processes so that each pattern has only one input and one output. Either strategy is applicable here.

Currently, this research has not focused on how to provide a way of **extending** the set of existing patterns. There has been other work in this area, such as DPnDP [11]. We hope to use and continue the work started by this system.

Our architecture addresses the problems of **openness** and **correctness**. The pattern layer addresses correctness by generating template code from the design patterns. Since the generated code cannot be modified, we can strictly enforce the structure of the program. Combined with a run–time library, this code can offer a high–level model that frees the user from the low–level details of parallel programming and guarantees the correctness of the parallel structures.

The subsequent layers of the system are intended to address the problem of openness by providing successively more access to both the generated code and the run–time system. The intermediate layer provides more control over the user program by providing access to the generated code of the first layer, but still provides a high–level programming model for easier programming. The user can optimize the generated code, but is then responsible for its correctness. Finally, the last layer provides access to the complete programming system.

Programming This category evaluates the style and structure of the application code written by the user.

In examining the language characteristics, we should emphasize that the CO_2P_3S architecture is intended to be independent of language. As such, it should be possible to use the architecture for a variety of languages, satisfying the goal of using an existing, **familiar programming language**. The definition of this characteristic also includes the ideal of preserving the semantics and syntax of this programming language. We purposefully stray from this ideal, though, as we feel that preserving the semantics of a sequential language unnecessarily limits the potential concurrency. As a small example, consider programming languages that support run–time exceptions. To fully preserve the sequential semantics, it is necessary to execute every statement in order. However, concurrent execution could execute code that would not be executed in the event of an exception [14]. Without concurrent execution, computational parallelism is useless. A limited set of new keywords can provide a usable abstraction for parallel programming without greatly disturbing the rest of the language. We feel that the benefits of a well–planned abstraction that is properly integrated into the language can outweigh the risks of modifying a programming language. Nevertheless, we are only proposing these modifications at the intermediate code layer of our PPS with the understanding that at the native code layer, they will be translated to a standard programming language; in this case, Java or C++.

Finally, we anticipate that CO_2P_3S will fail to meet the **non–intrusiveness** characteristic. As noted by Singh *et al.*, the only way to fully address this problem is to implement a compiler that automatically generates a parallel program from sequential code (which also solves the language objective). Unfortunately, current compiler technology cannot create coarse–grained parallel programs.

User Satisfaction This last category focuses on a combination of performance and usability concerns.

Without an implemented and mature system, most of the characteristics in this last category cannot be evaluated. In particular, we cannot address the **support** and **usability** objectives. The **performance** objective is dealt with by our architecture for incremental tuning. **Portability** can be addressed by providing different implementations of the Native Code layer for each architecture.

5 Related Work

In addition to FrameWorks and Enterprise, there are several other graphical parallel programming systems. Mentat [5] and HeNCE [2] both represent programs as directed graphs. The programming model of Mentat is similar to that of Enterprise with some extensions for C++. However, neither system is pattern–based and neither supports incremental tuning. The parallel programming language P^3L [1] provides a set of design patterns that are composed to create programs. The programming model involves explicit communication that is type–checked at compile–time. Unfortunately, new languages impose a steep learning curve on new users. Also, the language is not object–oriented and does not support multiple abstractions. The DPnDP system mentioned in Section 4 is similar to the Mentat system except that the nodes in the graph may be implemented using design patterns. This project also provides an interface for adding new patterns.

There has also been some work done in verifying code based on design patterns. Sefika *et al.* [8] proposed a model for verifying a program's adherence to a design pattern using a combination of static and dynamic program information, and also suggest generating code from a pattern. The generation of code from a design pattern has been done for the patterns in [4] by Budinsky *et al.* [3] This project delivers the source code that implements each design pattern. This code is then modified by the user for its intended application. In contrast, we do not allow the user to immediately edit the generated code, so we can more rigourously enforce the constraints of the selected pattern.

6 Preliminary Results and Current Status

We have some preliminary results based on a hand–coded implementation of a mesh similar to that in Section 3. The example program we use is a reaction–diffusion texture generator [13]. The algorithm can be described as two interacting LaPlace equations, which simulate the reaction and diffusion of two chemicals over a surface. The result is a texture map that approximates zebra stripes. We solve the problem using straightforward convolution.

The program was executed on a SUN Ultra–SPARC 2 with 2 200MHz SuperSPARC processors and 128M of main memory. The program was a hand written Java program based that implements the mesh pattern described in this paper. We did not perform any optimizations on the communication structure itself, although we made some optimizations at the user code layer to reduce the amount of state moved between threads. The sequential program is a modified version of the mesh code tuned for the case where there is only one mesh element. The threaded version was run using native threads (multiple threads using both processors) and achieved a wall clock speed–up of 1.26. Although this speed–up is not very good, Java native thread interpreters are new technology and it should improve in the future.

We are currently implementing CO_2P_3S in Java. However, we are being careful to keep the architecture and model independent of any specific language. We will support a rich set of patterns for the first layer of the system, including meshes, trees, pipelines, master/slave, and iterators. The target architecture is a multiprocessor machine with shared memory executing threads in parallel. This is currently supported for Solaris 2.5.1. We may also investigate using networks of workstations with a message–passing package such as Voyager [6].

References

1. B. Bacci *et al.* P^3L: A structured high level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.
2. A. Beguelin *et al.* HeNCE: A hetergeneous network computing environment. Technical Report UT-CS-93-205, University of Tennessee, August 1993.
3. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley, Reading, Mass., 1994.
5. A. S. Grimshaw. Easy to use object–oriented parallel programming with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.
6. ObjectSpace, Inc. *Voyager Version 1.0 Beta 2.1.* http://www.objectspace.com.
7. J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel & Distributed Tech.*, 1(3):85–96, 1993.
8. M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high–level design models. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pages 387–396, March 1996.
9. 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, January 1991.
10. A. Singh, J. Schaeffer, and D. Szafron. Experience with template–based parallel programming. *Concurrency: Practice and Experience*, 1997. To appear.
11. 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)*, August 1996.
12. D. Szafron and J. Schaeffer. An experiment to measure the usability of parallel programming systems. *Concurrency: Practice and Experience*, 8(2):147–166, 1996.
13. Andrew Witkin and Michael Kass. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proccedings)*, 25(4):299–308, July 1991.
14. A. Zubiri. An assessment of Java/RMI for object–oriented parallelism. Master's thesis, Department of Computing Science, University of Alberta, 1997.