Generating Parallel Program Frameworks from Parallel Design Patterns

Steve MacDonald, Duane Szafron, Jonathan Schaeffer, and Steven Bromling

Department of Computing Science, University of Alberta, CANADA T6G 2H1
{stevem,duane,jonathan,bromling}@cs.ualberta.ca

Abstract. Object-oriented programming, design patterns, and frameworks are abstraction techniques that have been used to reduce the complexity of sequential programming. The CO_2P_3S parallel programming system provides a layered development process that applies these three techniques to the more difficult domain of parallel programming. The system generates correct frameworks from pattern template specifications at the highest layer and provides performance tuning opportunities at lower layers. Each of these features is a solution to a major problem with current parallel programming systems. This paper describes CO_2P_3S and its highest level of abstraction using an example program to demonstrate the programming model and one of the supported pattern templates. Our results show that a programmer using the system can quickly generate a correct parallel structure. Further, applications built using these structures provide good speedups for a small amount of development effort.

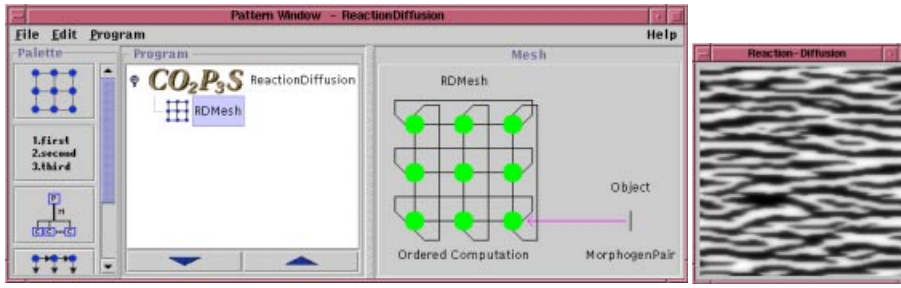
1 Introduction

Parallel programming offers substantial performance improvements to computationally intensive problems from fields such as computational biology, physics, chemistry, and computer graphics. Some of these problems require hours, days, or even weeks of computing time. However, using multiple processors effectively requires the creation of highly concurrent algorithms. These algorithms must then be implemented correctly and efficiently. This task is difficult, and usually falls on a small number of experts.

To simplify this task, we turn to abstraction techniques and development tools. From sequential programming, we note that the use of abstraction techniques such as object-oriented programming, design patterns, and frameworks reduces the software development effort. Object–oriented programming has proven successful through techniques such as encapsulation and code reuse. Design patterns document solutions to recurring design problems that can be applied in a variety of contexts [1]. Frameworks provide a set of classes that implement the basic structure of a particular kind of application, which are composed and specialized by a programmer to quickly create complete applications [2]. A development tool, such as a parallel programming system, can provide a complete toolset to support the development, debugging, and performance tuning stages of parallel programming.

The CO_2P_3S parallel programming system (Correct Object-Oriented Pattern-based Parallel Programming System, or "cops") combines the three abstraction techniques using a layered programming model that supports both the fast development of parallel

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 95–104, 2000.



(a) A screenshot of the Mesh template in CO_2P_3S. (b) Output image.

Fig. 1. The reaction–diffusion example in CO_2P_3S with an example texture.

programs and the ability to tune the resulting programs for performance [3,4]. The highest level of abstraction in CO_2P_3S emphasizes correctness by generating parallel structural code for an application based on a pattern description of the structure. The lower layers emphasize openness [7], gradually exposing the implementation details of the generated code to introduce opportunities for performance debugging. Users can select an appropriate layer of abstraction based on their needs.

This approach advances the state of the art in pattern-based parallel programming systems research by providing a solution to two recurring problems. First, CO_2P_3S generates correct parallel structural code for the user based on a pattern description of the program. In contrast, current pattern-based systems also require a pattern description but then rely on the user to provide application code that matches the selected structure. Second, the openness provided by the lower layers of CO_2P_3S gives the user the ability to tune an application in a structured way. Most systems restrict the user to the provided programming model and provide no facility for performance improvements. Those systems that do provide openness typically strip away all abstractions in the programming model immediately, overwhelming the user with details of the run-time system.

CO_2P_3S provides three layers of abstraction: the Patterns Layer, the Intermediate Code Layer, and the Native Code Layer. The Patterns Layer supports pattern-based parallel program development through framework generation. The user expresses the concurrency in a program by manipulating graphical representations of parallel design pattern templates. A template is a design pattern that is customized for the application via template parameters supplied by the user through the user interface. From the pattern specification, CO_2P_3S generates a framework implementing the selected parallel structure. The user fills in the application–dependent parts of the framework to implement a program. The two remaining layers, the Intermediate Code Layer and the Native Code Layer, allow users to modify the structure and implementation of the generated framework for performance tuning. More details on CO_2P_3S can be found in [3,4].

In this paper, we highlight the development model and user interface of CO_2P_3S using an example problem. CO_2P_3S is implemented in Java and creates multithreaded parallel frameworks that execute on shared memory systems using a native–threaded JVM that allows threads to be mapped to different processors. Our example is a reaction–

diffusion texture generation program that performs a chemical simulation to generate images resembling zebra stripes, shown in Figure 1. This program uses a Mesh pattern template which is an example of a parallel structural pattern for the SPMD model. We discuss the development process and the performance of this program. We also briefly discuss two other patterns supported by CO_2P_3S and another example problem. Our results show that the Patterns Layer is capable of quickly producing parallel programs that obtain performance gains.

2 Reaction–Diffusion Texture Generation

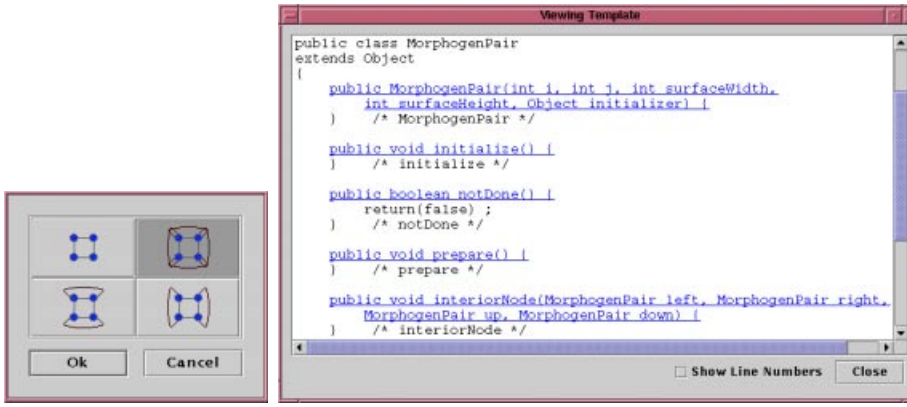
This section describes an example program that uses one of the CO_2P_3S pattern templates. The goal is to show how CO_2P_3S simplifies the task of parallel programming by generating correct framework code from a pattern template. This allows a user to write only sequential code to implement a parallel program. To accomplish this goal, a considerable amount of detail is given about the pattern template, its parameters, and the framework code that is generated.

Reaction–diffusion texture generation simulates two chemicals called *morphogens* as they simultaneously diffuse over a two–dimensional surface and react with one another [8]. This simulation, starting with random concentrations of each morphogen across the surface, can produce texture maps that approximate zebra stripes, as shown in Figure 1(b). This problem is solved using convolution. The simulation executes until the change in concentration for both morphogens at every point on the surface falls below a threshold. This implementation allows the diffusion of the morphogens to wrap around the edges of the surface. The resulting texture map can be tiled on a larger display without any noticeable edges between tiles.

2.1 Design Pattern Selection

The first step in implementing a pattern-based program is to analyze the problem and select the appropriate set of design patterns. This process still represents the bottleneck in the design of any program. We do not address pattern selection in this paper, but one approach is discussed in [5]. Given our problem, the two–dimensional Mesh pattern is a good choice. The problem is an iterative algorithm executed over the elements of a two–dimensional surface. The concentration of an element depends only on its current concentration and the current concentrations of its neighbouring elements. These computations can be done concurrently, as long as each element waits for its neighbours to finish before continuing with its next iteration.

Figure 1(a) shows a view of the reaction–diffusion program in CO_2P_3S. The user has selected the Mesh pattern template from the palette and has provided additional information (via dialog boxes such as that in Figure 2(a)) to specify the parameters for the template. The Mesh template requires the class name for the mesh object and the name of the mesh element class. The mesh object is responsible for properly executing the mesh computation, which the user defines by supplying the implementation of hook methods for the mesh element class. For this application, the user has indicated that the mesh object should be an instance of the `RDMesh` class and the mesh elements are



(a) Boundary conditions for the Mesh.

(b) Viewing template, showing default implementations of hook methods.

Fig. 2. Two dialogs from CO_2P_3S.

instances of the `MorphogenPair` class. The user has also specified that the `MorphogenPair` class has no user–defined superclass, so the class `Object` is used.

In addition to the class names, the user can also define parameters that affect the mesh computation itself. This example problem requires a fully–toroidal mesh, so the edges of the surface wrap around. The mesh computation considers the neighbours of mesh elements on the edges of the surface to be elements on the opposite edge, implementing the required topology. The user has selected this topology from the dialog in Figure 2(a), which also provides vertical–toroidal, horizontal–toroidal, and non–toroidal options for the topology. Further, this application requires an instance of the Mesh template that uses a four–point mesh, using the neighbours on the four compass points, as the morphogens diffuse horizontally and vertically. Alternatively, the user can select an eight–point mesh for problems that require data from all eight neighbouring elements. Finally, the new value for a morphogen in a given iteration is based on values computed in the previous iteration. Thus, the user must select an *ordered* mesh, which ensures that iterations are performed in lock step for all mesh elements. Alternatively, the user can select a *chaotic* mesh, where an element proceeds with its next iteration as soon as it can rather than waiting for its neighbours to finish. All of these options are available in the Mesh Pattern template through the CO_2P_3S user interface in Figure 1(a).

2.2 Generating and Using the Mesh Framework

Once the user has specified the parameters for the Mesh pattern template, CO_2P_3S uses the template to generate a framework of code implementing the structure for that specific version of the Mesh. This framework is a set of classes that implement the basic structure of a mesh computation, subject to the parameters for the Mesh pattern template. This structural framework defines the classes of the application and the flow of

control between the instances of these classes. The user does not add code directly to the framework, but rather creates subclasses of the framework classes to provide application–dependent implementations of "hook" methods. The framework provides the structure of the application and invokes these user–supplied hook methods. This is different than a library, where the user provides the structure of the application and a library provides utility routines. A framework provides design reuse by clearly separating the application–independent framework structure from the application–dependent code. The use of frameworks can reduce the effort required to build applications [2].

The Patterns Layer of CO_2P_3S emphasizes correctness. Generating the correct parallel structural code for a pattern template is only part of this effort. Once this structural code is created, CO_2P_3S also hides the structure of the framework so that it cannot be modified. This prevents users from introducing errors. Also, to ensure that users implement the correct hook methods, CO_2P_3S provides template viewers, shown in Figure 2(b), to display and edit these methods. At the Patterns Layer, the user can only implement these methods using the viewers. The user cannot modify the internals of the framework and introduce parallel structural errors. To further reduce the risk of programmer errors, the framework encapsulates all necessary synchronization for the provided parallel structure. The user does not need to include any synchronization or parallel code in the hook methods for the framework to operate correctly. The hook methods are normal, sequential code. These restrictions are relaxed in the lower layers of CO_2P_3S.

For the Mesh framework, the user must write two application–specific sections of code. The first part is the mainline method. A sample mainline is generated with the Mesh framework, but the user will likely need to modify the code to provide correct values for the application. The second part is the implementation of the mesh element class. The mesh element class defines the application–specific parts of the mesh computation: how to instantiate a mesh element, the mesh operation that the framework is parallelizing, the termination conditions for the computation, and a gather operation to collect the final results. The structural part of the Mesh framework creates a two–dimensional surface of these mesh elements and implements the flow of control through a parallel mesh computation. This structure uses the application–specific code supplied by the user for the specifics of the computation.

The mainline code is responsible for instantiating the mesh class and launching the computation. The mesh class is responsible for creating the surface of mesh elements, using the dimensions supplied by the user. The user can also supply an optional *initializer* object, providing additional state to the constructor for each mesh object. In this example, the initializer is a random number generator so that the morphogens can be initialized with random concentrations. Analogously, the user can also supply an optional *reducer* object to collect the final results of the computation, by applying this object to each mesh element after the mesh computation has finished. Once the mesh computation is complete, the results can be accessed through the reducer object. Finally, the user specifies the number of threads that should be used to perform the computation. The user specifies the number of horizontal and vertical blocks, decomposing the surface into smaller pieces. Each block is assigned to a different thread. This information is supplied at run-time so that the user can quickly experiment with different surface

```
import java.util.Random ;
public class MorphogenPair
{
    protected Morphogen morph1, morph2 ;

    public MorphogenPair(int x,int y,int width,int height,
                Object initializer) {
        Random gen = (Random) initializer;
        morph1 = new Morphogen(1.0-(gen.nextDouble()*2.0),2.0,1.0);
        morph2 = new Morphogen(1.0-(gen.nextDouble()*2.0),2.0,1.5);
    } /* MorphogenPair */

    public boolean notDone() {
        return(!(morph1.hasConverged() && morph2.hasConverged()));
    } /* notDone */

    public void prepare() {
        morph1.updateConcentrations();
        morph2.updateConcentrations();
    } /* prepare */

    public void interiorNode(MorphogenPair left, right, up, down) {
        morph1.simulate(left.getMorph1(),right.getMorph1(),
                up.getMorph1(),down.getMorph1(),morph2, 1);
        morph2.simulate(left.getMorph2(),right.getMorph2(),
                up.getMorph2(),down.getMorph2(),morph1, 2);
    } /* interiorNode */

    public void postProcess() {
        morph1.updateConcentrations();
    } /* postProcess */

    public void reduce(int x,int y,int width,int height,Object reducer) {
        Concentrations result = (Concentrations) reducer;
        result.concentration[x][y] = morph1.getConcentration() ;
    } /* reduce */
    // Define two accessors for the two morphogens.
} /* MorphogenPair */
```

Fig. 3. Selected parts of the `MorphogenPair` mesh element class.

sizes and numbers of threads. If these values were template parameters, the user would have to regenerate the framework and recompile the code for every new experiment.

The application–specific code in the Mesh framework is shown in Figure 3. The user writes an implementation of the mesh element class, `MorphogenPair`, defining methods for the specifics of the mesh computation for a single mesh element. When the framework is created, stubs for all of these methods are also generated. The framework iterates over the surface, invoking these methods for each mesh element at the appropriate time to execute the complete mesh computation. The sequence of method calls for the application is shown in Figure 4. This method is part of the structure of the Mesh framework, and is not available to the user at the Patterns Layer. However, this code shows the flow of control for a generic mesh computation. Using this code with the `MorphogenPair` implementation of Figure 3 shows the separation between the application–dependent and application–independent parts of the generated frameworks.

The constructor for the mesh element class (in Figure 3) creates a single element. The `x` and `y` arguments provide the location of the mesh element on the surface, which is of dimensions `width` by `height` (from the constructor for the mesh object). Pro-

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

Fig. 4. The main loop for each thread in the Mesh framework.

viding these arguments allows the construction of the mesh element to take its position into account if necessary. The constructor also accepts the initializer object, which is applied to the new mesh element. In this example, the initializer is a random number generator used to create morphogens with random initial concentrations.

The `initialize()` method is used for any initialization of the mesh elements that can be performed in parallel. In the reaction–diffusion application, no such initialization is required, so this method does not appear in Figure 3.

The `notDone()` method must return true if the element requires additional iterations to complete its calculation and false if the computation has completed for the element. Typical mesh computations iterate until the values in the mesh elements converge to a final solution. This requires that a mesh element remember both its current value and the value from the previous iteration. The reaction–diffusion problem also involves convergence, so each morphogen has instance variables for both its current concentration and the previous concentration. When the difference between these two values falls below a threshold, the element returns false. By default, the stub generated for this method returns false, indicating that the mesh computation has finished.

The `interiorNode(left,right,up,down)` method performs the mesh computation for the current element based on its value and the values of the supplied neighbouring elements. This method is invoked indirectly from the `operate()` method of Figure 4. There are, in fact, up to nine different operations that could be performed by a mesh element, based on the location of the element on the surface and the boundary conditions. These different operations have a different set of available neighbouring elements. For instance, two other operations are `topLeftCorner(right,down)` and `rightEdge(left,up,down)`. Stubs are generated for every one of the nine possible operations that are required by the boundary conditions selected by the user in the Mesh pattern template. For the reaction–diffusion example, the boundary conditions are fully–toroidal, so every element is considered an interior node (as each element has all four available neighbours since the edges of the surface wrap around). This method computes the new values for the concentrations of both of its morphogen objects, based on its own concentration and that of its neighbours.

The `prepare()` method performs whatever operations are necessary to prepare for the actual mesh element computation just described. When an element computes its new values, it depends on state from neighbouring elements. However, these elements may be concurrently computing new values. In some mesh computations, it is important that the elements supply the value that was computed during the previous iteration of the

computation, not the current one. Therefore, each element must maintain two copies of its value, one that is updated during an iteration and another that is read by neighbouring elements. We call these states the write and read states. When an element requests the state from a neighbour, it gets the read state. When an element updates its state, it updates the write state. Before the next iteration, the element must update its read state with the value in its write state. The `prepare()` method can be used for this update. The reaction–diffusion example uses a read and write state for the concentrations in the two morphogen objects, which are also used in the `notDone()` method to determine if the morphogen has converged to its final value.

The `postProcess()` method is used for any postprocessing of the mesh elements that can be performed in parallel. For this problem, we use this method to update the read states before the final results are gathered, so that the collected results will be the concentrations computed in the last iteration of the computation.

The `reduce(x,y,width,height,reducer)` method applies a reducer object to the mesh elements to obtain the final results of the computation. This reducer is typically a container object to gather the results so that they can be used after the computation has finished. In this application, the reducer is an object that contains an array for the final concentrations, which is used to display the final texture. Like the initializer, the reducer is passed as an `Object` and must be cast before it can be used.

2.3 The Implementation of the Mesh Framework

The user of a Mesh framework does not have to know anything about any other classes or methods. However, in this section we briefly describe the structure of the Mesh framework. This is useful from both a scientific standpoint and for any advanced user who wants to modify the framework code by working at the Intermediate Code Layer.

In general, the granularity of a mesh computation at an individual element is too small to justify a separate thread or process for that element. Therefore, the two–dimensional surface of the mesh is decomposed into a rectangular collection of block objects (where the number of blocks is specified by a user in the mesh object constructor). Each block object is assigned to a different thread to perform the mesh computations for the elements in that block. We obtain parallelism by allowing each thread to concurrently perform its local computations, subject to necessary synchronization. The code executed by each thread, for its block, is `meshMethod()` from Figure 4.

We now look at the method calls in `meshMethod()`. The `initialize()`, `prepare()`, and `postProcess()` methods iterate over their block and invoke the method with the same name on each mesh element. The `notDone()` method iterates over each element in its block, calling `notDone()`. Each thread locally reduces the value returned by its local block to determine if the computation for the block is complete. If any element returns true, the local computation has not completed. If all elements return false, the computation has finished. The threads then exchange these values to determine if the whole mesh computation has finished. Only when all threads have finished does the computation end. The `barrier()` invokes a barrier, causing all threads to finish preparing for the iteration before computing the new values for their block. The user does not implement any method for a mesh element corresponding to this method. Chaotic meshes do not include this synchronization. The `operate()`

Table 1. Speedups and wall clock times for the reaction–diffusion example.

	Processors	2	4	8	16
1680 by 1680	Speedup	1.75	3.13	4.92	6.50
	Time (sec)	5734	3008	1910	1448

method iterates over the mesh elements in the block, invoking the mesh operation method for that single element with the proper neighbouring elements as arguments. However, since some of the elements are interior elements and some are on the boundary of the mesh, there are up to nine different methods that could be invoked. The most common method is `interiorNode(left,right,up,down)`, but other methods may exist and may also be used, depending on the selected boundary conditions. The method is determined using a Strategy pattern [1] that is generated with the framework. Note that elements on the boundary of a block have neighbours that are in other blocks so they will invoke methods on elements in other blocks.

2.4 Evaluating the Mesh Framework

The performance of the reaction–diffusion example is shown in Table 1. These performance numbers are not necessarily the best that can be obtained. They are meant to show that for a little effort, it is possible to write a parallel program and quickly obtain speedups. Once we decide to use the Mesh pattern template, the structural code for the program is generated in a matter of minutes. Using existing sequential code, the remainder of the application can be implemented in several hours. To illustrate the relative effort required, we note that of the 696 lines of code for the parallel program, the user was responsible for 212 lines, about 30%. Of the 212 lines of user code, 158 lines, about 75%, was reused from the sequential version. We must point out, though, that these numbers are a function of the problem being solved, and not a function of the programming system. However, the generated code is a considerable portion of the overall total for the parallel program. Generating this code automatically reduces the effort needed to write parallel programs.

The program was run using a native threaded Java interpreter from SGI with optimizations and JIT turned on. The execution environment was an SGI Origin 2000 with 195MHz R10000 processors and 10GB of RAM. The virtual machine was started with 512MB of heap space. The speedups are based on wall clock times compared to a sequential implementation. These speedup numbers only include computation time.

From the table, we can see that the problem scales well up to four processors, but the speedup drops off considerably thereafter. The problem is granularity; as more processors are added, the amount of computation between barrier points decreases until synchronization is a limiting factor in performance. Larger computations, with either a larger surface or a more complex computation, yield better speedups.

3 Other Patterns in CO_2P_3S

In addition to the Mesh, CO_2P_3S supports several other pattern templates. Two of these are the Phases and the Distributor. The Phases template provides an extendible way

to create phased algorithms. Each phase can be parallelized individually, allowing the parallelism to change as the algorithm progresses. The Distributor template supports a data–parallel style of computation. Methods are invoked on a parent object, which forwards the same method to a fixed number of child objects, each executing in parallel.

We have composed these two patterns to implement the parallel sorting by regular sampling algorithm (PSRS) [6]. The details on the implementation of this program are in [4]. To summarize the results, the complete program was 1507 lines of code, with 669 lines (44%) written by the user. 212 of the 669 lines is taken from the JGL library. There was little code reuse from the sequential version of the problem as PSRS is an explicitly parallel algorithm. Because this algorithm does much less synchronization, it scales well up to 16 processors, obtaining a speedup of 11.2 on 16 processors.

4 Conclusions

This paper presented the graphical user interface of the CO_2P_3S parallel programming system. In particular, it showed the development of a reaction–diffusion texture generation program using the Mesh parallel design pattern template, using the facilities provided at the highest layer of abstraction in CO_2P_3S. Our experience suggests that we can quickly create a correct parallel structure that can be used to write a parallel program and obtain performance benefits.

Acknowledgements

This research was supported by grants and resources from the Natural Science and Engineering Research Council of Canada, MACI (Multimedia Advanced Computational Infrastructure), and the Alberta Research Council.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.
2. R. Johnson. Frameworks = (components + patterns). *CACM*, 40(10):39–42, October 1997.
3. S. MacDonald, J. Schaeffer, and D. Szafron. Pattern–based object–oriented parallel programming. In *Proceedings of ISCOPE'97*, LNCS volume 1343, pages 267–274, 1997.
4. S. MacDonald, D. Szafron, and J. Schaeffer. Object–oriented pattern–based parallel programming with automatically generated frameworks. In *Proceedings of COOTS'99*, pages 29–43, 1999.
5. B. Massingill, T. Mattson, and B. Sanders. A pattern language for parallel application programming. Technical Report CISE TR 99–009, University of Florida, 1999.
6. H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.
7. A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.
8. A. Witkin and M. Kass. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proccedings)*, 25(4):299–308, July 1991.