Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment

Kai Tan† Duane Szafron† Jonathan Schaeffer† John Anvik† Steve MacDonald‡

†Department of Computing Science, University of Alberta, Edmonton, AB, T6G 2E8, Canada

‡School of Computer Science, University of Waterloo, Waterloo, ON, N2L 3G1, Canada

{ cavalier, duane, jonathan, janvik }@cs.ualberta.ca, stevem@uwaterloo.ca

ABSTRACT

A design pattern is a mechanism for encapsulating the knowledge of experienced designers into a re-usable artifact. Parallel design patterns reflect commonly occurring parallel communication and synchronization structures. Our tools, CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System) and MetaCO_2P_3S, use *generative design patterns*. A programmer selects the parallel design patterns that are appropriate for an application, and then adapts the patterns for that specific application by selecting from a small set of code-configuration options. CO_2P_3S then generates a custom framework for the application that includes all of the structural code necessary for the application to run in parallel. The programmer is only required to write simple code that launches the application and to fill in some application-specific sequential hook routines. We use generative design patterns to take an application specification (parallel design patterns + sequential user code) and use it to generate parallel application code that achieves good performance in shared memory and distributed memory environments. Although our implementations are for Java, the approach we describe is tool and language independent. This paper describes generalizing CO_2P_3S to generate distributed-memory parallel solutions.

Categories and Subject Descriptors

D.1.3 [Concurrent Programming]: Distributed Programming, Parallel Programming; D.1.5 [Programming Techniques]: Object-Oriented Programming; D.2.11 [Sofware Architectures]: Patterns

General Terms

Performance, Design, Reliability, Languages

Keywords

Parallel Programming, Design Patterns, Frameworks, Programming Tools

1. INTRODUCTION

The past decade has seen enormous strides forward in software engineering methodologies and tools for developing sequential software. Many of these advances have quickly moved from academia to common practice, including object-oriented design, design patterns [17], and frameworks [34]. These technologies can lead to better program designs, fewer programming errors, and better code evolution.

In contrast, the parallel computing community has been slow to adopt new high-level programming techniques that have been successful in the sequential domain, even though several research tools support them:

- In practice few parallel applications are object-oriented, even though parallel object-oriented languages (such as Mentat [19] and Orca [2]) have existed for over a decade (POOMA [32] is an exception with a narrow focus).
- Design patterns for parallel programs have existed for two decades, in a variety of guises (e.g., skeletons [14][11], templates [36][38]). However, no widely-used parallel computing tool uses this technology.
- Frameworks have emerged as a powerful tool for rapid code development. To the best of our knowledge, there are no applications of this idea to building tools to support parallel code development.

The reality today is that the state of the art in parallel computing is represented by message-passing libraries (e.g., MPI [39]) and compiler directives (e.g., OpenMP [12]).

Many parallel programmers lose sight of the true cost of an application. It is, in part, a function of the program's execution time (faster is better), the number of times the program needs to be run (the amortization factor), and the cost of developing the code (real dollars). Many developers consider only the first two factors, while ignoring the third. Yet the reality is that for many applications, the rapid development and deployment of a correct parallel application is much more important than its execution time (at least in the short term). For example, dual-processor machines are ubiquitous, but most owners are not experienced parallel programmers. This class of users would like to build parallel applications with a minimum of effort; absolute speedup is often a secondary consideration.

There have been numerous attempts to develop high-level parallel programming tools that use abstraction to reduce parallel complexity so that users can quickly build structurally correct programs. Several tools require the user to write sequential stubs, with the tool inserting all the parallel code [33]. Despite these (often large) efforts, high-level parallel programming tools remain academic curiosities that are shunned by practitioners. There are three main reasons for this (others are explored in [37]):

1. **Performance**. Generic tools generally produce abstract parallel code with disappointing parallel performance. Even if such a tool were used to develop a first draft of a structurally correct implementation, most tools do not

PPoPP '03, June 11-13, 2003, San Diego, California, USA.

make the entire generated application code available in a usable form for incremental tuning.

2. **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.
3. **Architecture-independence**. Most tools are targeted to generate code for one type of parallel architecture. The type and amount of user input to the tool is dictated by the target architecture, and may not be reusable for different architectures. The ideal situation is to abstract the application parallelism from the architecture.

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

1.1 Design Patterns

Design patterns capture the knowledge of experienced object–oriented designers in a form that can be distributed to others [3]. Design patterns exist because designers do not solve all problems from first principles. In fact, experience plays a key role and a design pattern is an attempt to classify and describe this experience in small re-usable components.

Until recently, design patterns have been *descriptive*. They include a diagram and a description that consists of eleven parts: intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, and related patterns [17]. Patterns in this form are easy to distribute and are independent of software development tools, since they are only applied during application design. Experienced parallel programmers are familiar with the concept of design patterns, even if they haven't used them in this stylized form. For example, the well-known concepts of mesh, wavefront, master-slave, and pipeline can serve as the basis for parallel design patterns.

During design, each selected pattern must be *adapted* for use in the context of the current application, since each pattern is actually a family of solutions to similar problems. For example, the designer must choose between a four-point mesh and an eight-point mesh, and whether iteration will be Jacobi (synchronization between iterations) or Gauss-Seidel (no synchronization between iterations).

Traditionally, adapted patterns are the product of a design phase. During implementation, the adapted design pattern serves as a specification. Experienced programmers can transform the specification into code. In fact, experienced parallel programmers have probably implemented the master-slave pattern many times. Nevertheless, implementation involves writing code from scratch, using experience or sample design pattern code as a guide. This approach is a waste of valuable programmer time.

Recently, some researchers have tried to generate code from design patterns [5][8][13][15][28][45]. We have developed a *generative* design pattern (GDP) approach [25] that we have applied to parallel programs [22][24][26]. A *pattern designer* builds a GDP that contains pattern code-configuration options whose values can vary over well-defined domains. Each pattern also contains a set of code fragments for each valid set of code-configuration option settings. A programmer adapts a GDP for a particular application by specifying the values of pattern options and then pushing a button to generate code. The generated code is a framework whose code is configured on the basis of the options selected by the programmer. For example, in the case of the mesh design pattern, the generated parallel structure code for synchronized and unsynchronized meshes are quite different. However, from the user's point of view, this should be as simple as selecting an option to indicate which form of synchronization is desired.

1.2 Why Generative Design Patterns Work

We have previously shown how GDPs overcome the performance and generality obstacles for shared-memory applications using CO_2P_3S and $MetaCO_2P_3S$ [23]. There are three key reasons for this success. First, before generating code, the programmer selects code-configuration options that customize the design pattern template for a specific application. The existence of these options supports design patterns that are general enough for a wide range of applications, while facilitating the generation of non-generic code with good performance. Without using options before generation, the indiscriminate use of too many design patterns could cause significant performance problems [34].

However, one more step is often necessary to achieve high performance. Most high-level parallel programming tools use a programming model that suffers from a lack of "openness" [37]. The code generated by these tools is often difficult to tune for performance. In some cases, parts of the code are not available. In other cases, all the code is available, but it is not human readable. Too much effort may be required to understand the underlying software architecture.

The second key feature of GDPs is their ability to construct an open and layered programming model. All of the generated code is available to the programmer, 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 exposed. This approach provides performance gains that are commensurate with effort.

The third key to the generality and performance of GDPs is a tool to edit and create them. For pattern-based programming, current academic tools only support a small number of patterns and, with only a few exceptions, do not support the creation of new patterns. We have shown how to create a graphical GDP editor called $MetaCO_2P_3S$ [7] that supports generality by allowing a pattern designer to both edit GDPs and to create new ones. New GDPs must be first class in that they are just as general and powerful as the GDPs that are pre-loaded into tools like CO_2P_3S. We have also proposed a tool-independent XML format [6] for GDPs so that they can be used in any GDP-based parallel programming system. Generative design patterns reduce the time needed to implement parallel applications, are less prone to programmer error, promote rapid prototyping and code reuse, support performance tuning, and provide better overall software engineering benefits.

1.3 Generative Design Patterns for Distributed Memory Applications

Until now, CO_2P_3S only supported creating multi-threaded applications that executed on a shared-memory (SM) computer [23]. However, architecture-independence is a critical issue in

the utility of any parallel programming tool. Without support for distributed-memory (DM) applications, CO_2P_3S would be ineffective for a large user community. Ideally, one would like a tool to support a "design once, run anywhere" philosophy. Given an application specification (parallel design patterns + sequential user code), the user would like to be able to select an option that specifies the target architecture, and have the tool do the rest. In this paper, we describe how CO_2P_3S has been extended to support this capability.

There are several problems that need to be solved before GDPs can be used to generate DM code. In terms of generality, the goal is to ensure that a programmer can transparently use the same set of GDPs to generate code for SM computations and DM computations. In addition, the application-specific code supplied by the programmer must be the same in both cases. In other words, the only difference in writing a SM or DM program should be to push a different "architecture configuration" button. However, another generality issue is that it must also be possible for a pattern designer to rapidly edit and create GDPs that generate DM code.

On the performance side, high performance is much harder to obtain using DM. However, although our speedup expectations are lower, we must still be able to generate DM code whose performance is commensurate with effort. To attain this goal a DM GDP system must provide the following facilities:

1. an infrastructure to construct the whole distributed system,
2. a communication subsystem, and
3. a synchronization mechanism.

In addition, we need new versions of GDPs that transparently generate DM code that can use these facilities.

1.4 Research Contributions

The major research contributions of this paper are:

1. A demonstration that generative design patterns can be used to generate distributed-memory code that runs on a network of workstations. The programmer must only select the appropriate code-configuration options and supply application-specific sequential hook methods.
2. The only difference that a programmer sees in generating DM or SM code is to select a different option, along with a few statements during program launch that handle exceptions that can occur in the DM case, but not in the SM case.
3. Although performance is not spectacular, it is commensurate with effort and performance tuning is supported. The generated Java code uses standard widely-available network infrastructure software (Jini, RMI), with only straightforward modifications to enhance its performance. All the generated code is exposed to the user in an understandable format. This facilitates incremental tuning of the generated code.
4. Given the infrastructure we have created, it is straightforward to use an existing GDP editor, like MetaCO_2P_3S, to edit existing GDP patterns so that they can generate DM code using our infrastructure.

In Section 2, we describe the process for using a GDP to build a mesh application that runs on a network of workstations. We also provide performance results and illustrate performance tuning. In Section 3 we briefly describe two other applications that use three other GDPs. In Section 4, we describe the parallel architecture of our distributed-memory implementations. We used widely available infrastructure: Jini, RMI and JavaSpaces. In Section 5, we outline some enhancements we made to this infrastructure. In Section 6, we list our conclusions and discuss the future of GDPs.

2. USING GENERATIVE DESIGN PATTERNS TO GENERATE DISTRIBUTED MEMORY MESH CODE

In this section, we illustrate the process of using a GDP to generate DM Java code for a mesh application using CO_2P_3S. Some may argue that using Java for high-performance numerical computations is ill-advised. However, progress is being made in using Java for such applications [4][29] and as researchers, we must anticipate future new parallel processing trends.

Many computer simulation and animation applications model a surface as a two-dimensional mesh. For example, in image processing, the image is a two-dimensional mesh, where each mesh node represents one pixel. Reaction-diffusion is a chemical process that simulates graphical patterns. Two or more chemicals diffuse over the surface and react to form a stable pattern. In computer graphics, reaction-diffusion simulation can be used to generate a zebra stripe texture.

The design process is called the Parallel Design Pattern (PDP) process and it has five steps [22]:

1. Identify one or more design patterns for the parallel parts of the application.[1]
2. Adapt the chosen pattern templates. A programmer selects application-specific code-configuration options, allowing them to customize the design patterns to their application.
3. A framework is generated with application-dependent sequential hook methods. The programmer fills in the hook methods with application-specific code to obtain a correct functional parallel program (modulo any bugs the user introduced in the hook methods).
4. Evaluate the resulting application for initial performance results. If they are not acceptable, inspect the generated framework code and make performance-improving changes.
5. Re-evaluate the performance of the modified application. If it is still not acceptable, return to Step 4 or, in some cases, return to Step 1 (some applications are amenable to different parallel design patterns).

For the reaction-diffusion application, we can verify that the mesh pattern can be applied since the algorithm defines a two-dimensional rectangular surface consisting of chemicals, each of which reacts with immediate neighbors and changes over time. Figure 1 shows a mesh design pattern in CO_2P_3S. The left pane shows selectable GDPs and the center pane shows a mesh pattern that has been selected for this application. This completes Step 1. One approach to selecting an appropriate pattern is to use a pattern language [27].

[1] CO_2P_3S has also been used to support design patterns for sequential programming [25].

The mesh pattern has six code-configuration options. For this application, the options and their settings are (Step 2):

- The *mesh class name* is **RDMesh**.
- The *mesh element class name* is **MorphogenPair**.
- The *superclass of the mesh element class* (MorphogenPair) is **Object**.
- The *topology of the mesh* is **fully-toroidal**.
- The *number of neighbors* is **4**.
- The *mesh ordering* is set to **ordered**.

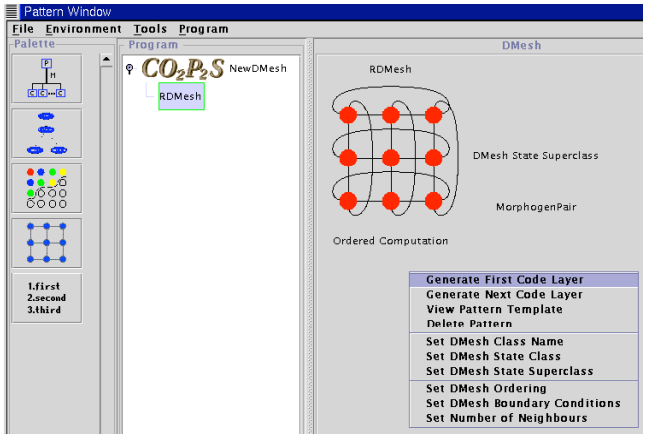

Figure 1. Using a mesh design pattern in CO_2P_3S

The programmer sets the options using various dialog boxes, and the results are shown in the right pane of Figure 1. Each combination of code-configuration options ends up generating a different parallel framework.

Step 3 of the PDP process is to generate framework code and fill in the sequential hook methods. The code for the reaction-diffusion application partitions the mesh over multiple machines. The main execution loop for each partition is shown in Figure 2. Each of the partition methods: `notDone()`, `preProcess()`, `prepare()`, `postProcess()`, and `operate()` contains a loop that iterates over all mesh elements in the partition and invokes an appropriate hook method on each in turn. The partition methods, `notDone()` and `barrier()` require global synchronization and will be discussed in Section 3 along with the technique used to exchange border elements and other data between machines.

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

Figure 2. The main execution loop for each mesh partition

The programmer does not implement the partition methods since their method bodies are generated. The programmer only implements the sequential hook methods that operate on individual mesh elements. These sequential hook methods are shown in Figure 3. In this case, the mesh surface is fully toroidal, so all mesh elements are updated the same way. If the topology was non-toroidal, additional `operate()` methods such as the one in Figure 4, would be generated for special positions on the mesh surface. The programmer does not have to write code that calls the appropriate operate method. That code is generated as part of the framework. The programmer only has to implement the appropriate operate methods for individual kinds of mesh elements.

```
MorphogenPair(int i, int j, int surfaceWidth,
              int surfaceHeight, Object initializer){}
(* This constructor method uses the user-specified
   initializer object to initialize the mesh node at
   (i,j) of the surface. *)

void preProcess() {}
void prepare() {}
void postProcess() {}
(* These three methods allow users to specify customized
   code fragments for a mesh element at different points
   of the execution loop. *)

boolean notDone() {}
(* Termination condition for an individual element *)

void reduce(int i, int j, int surfaceWidth,
            int surfaceHeight, Object reducer) {}
(* Apply the user-supplied reducer object to the current
   element to gather the results of the computation. *)

void operate(MorphogenPair north, MorphogenPair east,
             MorphogenPair south, MorphogenPair west) {}
(* This method defines how the current MorphogenPair
   reacts with its immediate neighbors. *)
```

Figure 3. The sequential hook methods for the mesh elements

```
operateLeftEdge(MorphogenPair right, MorphogenPair up,
                                      MorphogenPair down)
(* Used to update mesh elements on the left edge. *)
```

Figure 4. A mesh update method for a non-toroidal mesh

To complete the application (Step 3 of the PDP process), the programmer must write some simple code to launch the application. The launch code for the distributed-memory version of the reaction-diffusion application is shown in Figure 5. The shared-memory version is the same, except that it does not use the try-catch clause. Instead, it consists only of the code in the try-block. This is the only place that the programmer-supplied code is different between the distributed-memory and shared-memory options.

```
try {
   mesh = new RDMesh(surfaceWidth, surfaceHeight,
              meshWidth, meshHeight, initializer, reducer);
   mesh.launch();
}
catch(java.rmi.RemoteException remoteException)
{
   remoteException.printStackTrace();
}
```

Figure 5. Launching the reaction-diffusion application

At this stage, we now have a completely functional and correct program (modulo any bugs the user has introduced in the hook methods). The program can be run and its correctness can be verified. Only now with a correct program in hand do we consider whether the performance is satisfactory. By generating all the parallel structure code, CO_2P_3S eliminates most sources of concurrent programming errors. This reduces the code development time, and increases the user's confidence in the correctness of their solution.

Step 4 in the PDP process is to run the program and evaluate its performance. The central JVM for the main program was started with a 512MB heap space. The distributed JVMs were started with a 256MB heap space. The speedups are based on the average wall clock time for ten executions compared to the sequential execution time using a HotSpot virtual machine. Note that the timings consider only the computation time and boundary-exchange time; the initialization and result gathering times are not included. Table 1 contains the results.

The results given in Table 1 are not very satisfying, especially when compared to the results obtained from a CO_2P_3S-generated shared-memory implementation, which achieves speedups of 3.7 on 4 processors for the same problem. However, to obtain the distributed-memory program for the reaction-diffusion application using CO_2P_3S was easy: change the memory code-configuration option from *shared-memory* to *distributed-memory,* push the *generate code* button and add the exception-handling code for the launch.

Table 1. Initial performance for a distributed-memory implementation of the reaction-diffusion application (seconds)

Mesh Size	1 processor	4 processors	Speedup
400×400	31	103	0.3
800×800	120	110	1.1
1200×1200	267	131	2.0

Since the distributed-memory version's performance is not satisfactory, we can try to improve it by examining the generated code. Using the real-time monitoring function provided in CO_2P_3S, we discovered that each participating processor was less than 40 percent utilized when the mesh size was less than 1200×1200. Most of the time was wasted on synchronization and boundary exchange. As the mesh size rose to 1200×1200, each processor received more work to do, enough to offset the high communication overhead. We realized that we would have to examine the generated code and look for opportunities to remove synchronizations and/or communications.

Consider the main execution loop in the framework code of Figure 2. All processes need to synchronize by calling the `barrier()` method twice for each iteration. One call is the explicit call to `barrier()` that appears between the `prepare()` method and the `operate()` method. This synchronization is necessary to ensure that the computation is ordered (Jacobi iteration). The other call to `barrier()` is a hidden one inside the `notDone()` method that can be easily found by inspecting the generated framework code. This synchronization is necessary since all processes must submit their own view of whether they are done or not, before a final decision is made whether to terminate the entire computation or continue iterating.

A distributed barrier is very expensive since it involves passing multiple messages between all the mesh partitions. If we can reduce the number of barrier synchronizations by one for each iteration, it will be a huge performance win, since it will reduce the total synchronization overhead by 50%.

On inspection, it was discovered that the code in the `prepare()` method can be incorporated into the `notDone()` method before the synchronization point already in that method. This results in the hidden barrier call at the end of the `notDone()` method being adjacent to the explicit `barrier()` invocation in the loop. Since the two are adjacent, one is redundant and the explicit barrier invocation can be removed. The resulting code is shown in Figure 6.

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

Figure 6. The modified execution loop for mesh partitions

After modifying the generated code, we re-ran the experiments and the results are shown in Table 2. This true anecdote is a good illustration of how performance tuning can work when coupled with the readable parallel structure code that is generated by CO_2P_3S.

Table 2. Tuned performance for a parallel mesh application

Mesh Size	1 processor	4 processors	Speedup
400×400	31	64	0.5
800×800	120	70	1.7
1200×1200	267	91	2.9

Given how well this optimization worked in the reaction-diffusion equation, we quickly realized that we could go back to the GDP for the mesh and modify the generated code so that the code would be in the form of Figure 6 instead of the form of Figure 2. However, there is a better solution. If the `prepare()` code is always executed before the rest of the code in `notDone()`, this code may be executed one more time than is necessary, since if the program is done, the preparation was unnecessary the last time through the loop. This will not be a concern unless the preparation time actually increases as the computation proceeds, but this situation can occur. This kind of scenario is common in parallel computing and the GDPs have a great solution for it. A new code-configuration option can be added to the GDP that differentiates between these two cases; the code that is generated will depend on the setting of this option. Such an option is called a *performance option* since the programmer will often try different settings to determine the most efficient one for the particular application.

The last step in the PDP process (Step 5) is to evaluate whether more performance tuning is required. If so, additional changes can be made until the desired performance level is reached. For

example, CO_2P_3S automatically does data partitioning across the processors and decides on the granularity of the computations. The CO_2P_3S default settings may be sub-optimal for a given application. With a mesh application, CO_2P_3S, simply divides the mesh evenly among the number of processors available as specified by the programmer when instantiating the mesh using arguments, `meshWidth` and `meshHeight`, from Figure 5. However, in other patterns, like a distributor pattern, a method call is made with an array argument and this array is distributed by making the same method call to several processes with the array distributed using one of: *pass through*, *block*, *striped* or *neighbor distribution* [22]. The user can experiment with different values for this performance parameter. The changes needed to experiment with different data partitions and/or granularity are well documented in the generated code.

3. GENERATING DISTRIBUTED MEMORY CODE FOR THE WAVEFRONT PATTERN

In this section we report the results of using GDPs to generate DM code for another application. We present a biological sequence alignment application that uses the wavefront GDP and a sorting application that uses the composition of two GDPs, the phases and distributor patterns.

The biological sequence alignment problem can be solved using a dynamic programming matrix, which can be modeled by the wavefront pattern [1]. In essence, aligning two sequences with lengths of m and n reduces to finding a maximum cost path through a matrix of size $(m+1)\times(n+1)$. An extra row and column are added to represent initial scores of the matrix to start the computation. The computation starts from the top left corner and proceeds to the bottom right corner. The resulting paths across the matrix represent different combinations of the possible operations: letters are matched, mismatched, and matched with gaps inserted into the original sequences.

This application is an example of a wavefront computation since each element is dependent on some of its neighbors' values, and the computation ripples across the matrix like a wavefront. In this application, the value of each element depends on the neighboring elements to the north (above), west (left) and northwest (above and to the left). However, other applications may have other dependency options.

The wavefront code-configuration option selections for this application are:

- The *wavefront element class name* is **SAElement**.
- The *type of wavefront elements* is **int**.
- The *matrix shape* is **full** matrix.
- The *dependency set* is **north**, **northwest** and **west**.
- The *immediate neighbors only* option is **true**.
- The *notification technique* is **Push**.

A CO_2P_3S screenshot of this application is shown in Figure 7. The class representing individual wavefront elements is called **SAElement**. However, since this application only needs to store integers in the matrix, the type of wavefront element is marked as **int**. In cases like this, CO_2P_3S generates code that uses a two-dimensional array of ints (instead of an array of objects) and static methods are generated to update the wavefront elements (instead of using instance methods). This approach increases performance considerably. For this application, a **full** matrix is needed. Other option choices are a banded matrix or a triangular matrix. The dependency set for this application is **north**, **west** and **northwest**. Not all choices are legal, since if two opposite directions are selected, the computation will deadlock. The CO_2P_3S user interface prevents the programmer from selecting illegal option values.

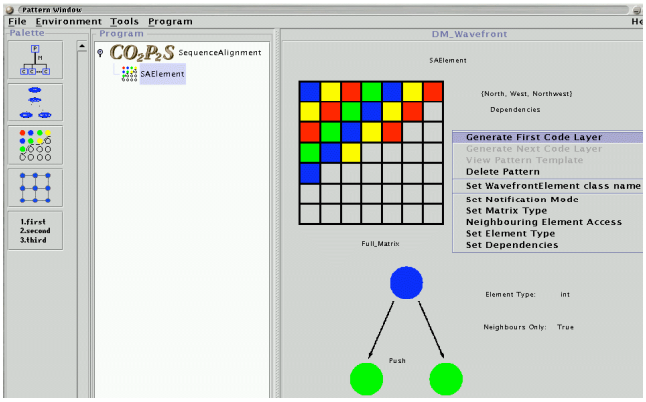

Figure 7. Using a wavefront design pattern in CO_2P_3S

The immediate neighbors option specifies whether an element needs all the elements in particular directions, or only the immediate neighbors in those directions. In this application, only the immediate neighbors are required so the option is set to **true**. We have implemented other applications, like a skyline matrix application and a matrix product chain application [1], where this option must be set to **false**.

The last option is a performance option. Performance options don't affect the hook methods that the user must write, only the performance of the generated application. Even though the hook methods are consistent across all choices of performance options, the internal structure code of the framework may vary radically. The notification method specifies whether an element should poll the elements it is dependent on (**Pull**) to see whether their computations are complete, or whether an element should inform its dependents when its own computation is complete (**Push**). For this application, the performance of both options is similar.

The sequential hook methods for the wavefront pattern are similar to the ones for the mesh from Section 2. However, in this case, there are many more `operate()` methods since we need custom ones for the top row, left column and top-left corner, as well as the general one for the interior.

The parallel implementation of a wavefront divides the matrix into blocks and performs the computation on multiple blocks concurrently. An algorithm for the main execution loop is shown in Figure 8.

```
while (computation not done) {
   Get one block from the controller's worklist
   Process this block
   Notify the dependent blocks
   Send the boundaries
}
```

Figure 8. The execution loop for each wavefront processor

A controller maintains a work-list of blocks whose dependencies have been satisfied. When a processor is idle, it

takes a block from the controller's work-list, processes the block, notifies the dependent blocks that this block is done, and sends the boundary values to the controller so that they can be used to update other blocks. Of course, all of this is in the code generated by CO_2P_3S and the user does not need to know any of these details to build a correct parallel implementation.

Table 3 lists the time and the speedup of the sequence alignment application using the wavefront pattern and two sequences of size 10,000. The system configuration is the same as described in Section 2. The speedup rises slowly as more processors are added. Although these speedup numbers are not great, they still indicate some speedup in exchange for quick parallelization. For this application, we did not try to tune the performance by modifying the generated code.

Table 3. Performance results for a wavefront application (milliseconds)

Processors	1	2	4	6	8
Time	6,383	7,321	3,733	2,633	2,115
Speedup	-	0.9	1.7	2.4	3.0

Besides the mesh and wavefront, we have also added distributed-memory code generation to the phases and distributor patterns, and implemented a parallel sorting application, called PSRS [35], to test them. The phases pattern is used to break a parallel computation up into distinct phases with synchronization at the end of each phase. The distributor is a master-slave pattern that performs a computation on an array argument, by sending parts of the array to different processors. It has a configuration-option that selects between various distributions like block, striped and neighbors. Details can be found in [44]. Speedups ranged from 1.5 for two processors to 4.6 for eight processors. However, the magnitude of the speedup is not important. We showed again, that a GDP could be modified to generate distributed-memory code, using two design patterns that had previously been created to generate shared-memory code. In addition, the generated distributed-memory code has a reasonable performance improvement that was commensurate with the parallelization effort.

4. THE PARALLEL ARCHITECTURE TO SUPPORT DISTRIBUTED MEMORY CODE

In this section, we describe our parallel architecture that uses Jini, RMI and JavaSpaces. We describe:

- an infrastructure to construct the distributed system,
- a communication subsystem, and
- a synchronization mechanism.

Since CO_2P_3S supports Java, we have used two Java-centric infrastructure technologies, Jini [42] and RMI [41], to design the system architecture. In addition to a standard API, Jini provides extensive support for Java-based distributed computing. Jini can be used to build a distributed system with a scalable and dynamically-configurable architecture. It also provides efficient process coordination mechanisms, a central object space (a JavaSpaces service) for global synchronization and data sharing, a customizable communication scheme, and a lease renewal service to manage communication failures. These facilities reduced the complexities of constructing our distributed architecture and allowed us to focus on creating parallel design patterns and on performance enhancements.

All system control tasks such as process spawning, process killing, synchronization, communication, and real-time performance monitoring are implemented using Jini technology and Java Native Interface (JNI) [43]. The generated application code contains multiple processes, where each is a Jini service running on a distributed machine. All Jini services locate each other through the Jini lookup service (LUS) and coordinate with each other directly or through a Jini Transaction Server and a JavaSpaces service. A JavaSpaces service [16] is an object-oriented version of the tuple-space that first appeared in the Linda system [9]. It is a central object space that provides synchronized access to the entries stored in it. A JavaSpaces service is implemented as a common Jini service for collaborative distributed applications.

4.1 CO_2P_3S Distributed Architecture and Infrastructure

The CO_2P_3S architecture for distributed-memory programs is shown in Figure 9. Each participating machine is shown as a dashed-line rounded rectangle with thick borders. The two participants at the bottom of the figure represent machines that perform the concurrent computations. Each of these computational machines contains a set of Jini services that interact with remote Jini services. A central control process, represented by the participant in the middle of the figure, provides a graphical user interface (GUI) that augments the GUI from the shared-memory version of CO_2P_3S. The GUI includes a tool menu with three options. One option allows the user to configure the environment by adding and removing machines for the computation. The second option launches all the Jini infrastructure components on the selected machines. The third option destroys all the Jini components and shuts down the environment. The participant at the top of the figure provides Jini resources for the computation, but do not perform the computations directly.

A generated application consists of a main program and a series of distributed slave processes. The main program is executed on the central control machine. Each process is implemented as a Jini activatable service [40] running on a distinct machine in the environment. Each slave process contains a proxy and a service. The Java activation system is used to make efficient use of resources on distributed machines. Rather than having each service process continually waiting for a request, each service contains a daemon process, called an *rmid*, which listens for requests and activates the appropriate service when needed. Each distributed machine, except the central control, runs an *rmid* to take care of the launching and shutting down of activatable processes.

One LUS is setup in the environment to control resource registration and discovery. The main program uses the LUS to obtain proxies for all distributed processes and to coordinate them. For instance, in the mesh application of Section 2, the main program creates a mesh data structure and divides it into smaller blocks. The process running the main program uses the LUS to find all registered processes that are able to process the blocks. The main program sends blocks to these processes via the proxies provided by the LUS.

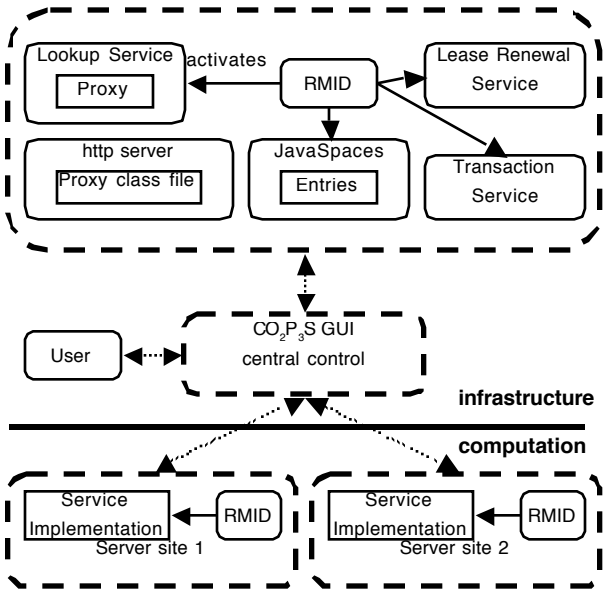

Figure 9. The architecture of distributed CO_2P_3S

4.2 CO_2P_3S Distributed Communication

Jini supports customizable communication. A service designer can choose any applicable protocol for communications between a service and its clients. Furthermore, the design choice for the service is totally transparent to the clients, who only need to load a proxy at runtime to discover the service's interface.

The default communication scheme used by Jini services is RMI, which is also used in all Jini infrastructure components such as the LUS, the JavaSpaces service, the transaction manager, and the lease renewal service. If RMI is used as the communication scheme in a user service, the proxy is a stub that is automatically generated by the RMI compiler (*rmic*). Using RMI greatly simplifies the communication design of distributed applications. However, since RMI is not open-ended and it hides almost all the communication details from the user, RMI is not very extensible or customizable. It leaves programmers little opportunity for fine-tuning in high-performance computing applications.

Java sockets are an alternative to RMI. In this case, a pattern designer would implement the communication details for the proxy and the service for each pattern. Sockets have more flexibility than RMI and may perform better. There are two reasons for this. First, more efficient wire protocols can be adopted. Second, the designer can choose to implement a thin proxy or a fat proxy [30]. A thin proxy simply forwards the request to the server and waits for a reply. A fat proxy can process some or all of a client's requests locally without sending them to the server. This approach is effective if a request can be more efficiently processed locally instead of involving network communications. This may occur due to the small granularity of a computation or the existence of local data or other resources. For example, a request to a remote server to display real-time graphics based on the client's input may be more efficient if the proxy processes the client's input directly and then draws the graphics locally.

We conducted an experiment to evaluate the performance difference between using RMI and Java sockets in the distributed CO_2P_3S environment. To conduct the performance test, we implemented a Jini service using RMI and then using a Java TCP socket. The test program contained three major participants, two services and a client, all of which run in the Jini environment. The performance comparison focused on the difference between the pure overhead involved in the coding and decoding of method invocation information of the RMI implementation and the TCP sockets implementation. Three methods were used, each with an argument array containing 1,000 elements. The element types were Object, String and Integer (not int). The semantics of the methods are not important. The test program was run on a PC Cluster with 19 nodes connected with a 100Mb Ethernet connection. Each node is a dual Athlon MP 1800+ CPU and 1.5GB of RAM. The operating system kernel is Linux 2.4.18-pfctr and the JDK version is Java HotSpot VM 1.3.1. The virtual machine was started with a 256MB heap space. Table 4 shows the average execution time (averaged over 100 runs). It was found that Java sockets are faster than RMI, and that the performance improvement varies considerably among different applications depending on the method arguments.

Table 4. Performance comparison of RMI versus sockets (milliseconds)

	Objects	Strings	Integers
RMI	36.5	4.6	6.3
Java sockets	26.6	4.1	3.5
% faster	37%	12%	80%

Communication is often a performance bottleneck in distributed computing. Customized communication (via sockets) may achieve higher performance by allowing the pattern designer to reduce communication overhead for specific patterns. However, our goal is not only to reduce the complexities of parallel programming in a distributed environment, but also to support the rapid design of new parallel patterns. The conflict between requirements for efficiency in communication and broad abstractions to assist pattern designers suggests that a compromise is necessary. We selected RMI to design all required Jini services because of its simplicity. However, we also modified the existing RMI implementation to diminish the performance gap between it and Java sockets. In particular, we created a modified version of RMI that uses a more compact and efficient serialization scheme designed specifically for high-performance computing. Using the modified RMI maintains performance while reducing the complexity of the design patterns. This is beneficial, both to the pattern designer who must construct the patterns and to the programmer who may need to tune them. Our RMI improvements are described in Section 5.

4.3 CO_2P_3S Distributed Synchronization

Parallel applications use synchronization mechanisms to keep shared data consistent and processes coordinated. One of the strengths of Java is that it supports multithreaded programming at the language level. Java provides monitors to fully support thread synchronization [46]. The Java Virtual Machine associates a monitor with each object and provides two opcodes (*monitorenter* and *monitorexit*) to access the

monitor lock. In Java, it is convenient to implement complex thread-level synchronization semantics based on monitors because in shared-memory systems, the heap space and the method area are shared among all threads.

We have already invested considerable effort in designing the shared-memory versions of GDPs using the Java Monitor programming model and we have many patterns whose code utilizes this model. Unfortunately, the Java RMI distributed-computing model has no direct support for the Monitor model. However, based on our current investment in this model, we decided to implement the Java Monitor in a distributed-memory environment [46][10]. This approach has four advantages:

1. It allowed us to use MetaCO_2P_3S to edit the SM versions of existing patterns to create DM versions quickly.
2. Parallel Java programmers are familiar with the Monitor programming model, so it is easier for them to write DM pattern implementations using this model.
3. A pattern designer who designs a new pattern can share code between the SM and DM versions of a pattern, since they share a similar syntax.
4. Programmers who want to tune the CO_2P_3S-generated code will find a familiar model to work with.

We used basic message passing to implement the monitor model. This implementation was self-contained and easy to use. However, because of the lack of lower-level communication support, setting up an all-to-all TCP connection between *N* processes was too expensive. UDP was no better since the lack of arrival guarantees and ordering guarantees made it difficult to correctly implement parallel semantics without additional checks at the application level.

Therefore, we re-implemented the Java monitor synchronization model using JavaSpaces technology. We created a JavaSpaces service to store variables that are shared among a collection of processes. Since the JavaSpaces service provides mutually exclusive access to the shared variables, we were able to implement a basic mutex lock very easily.

An object-diagram of the synchronization subsystem is shown in Figure 10. Our synchronization package actually includes the following classes: Barrier, Monitor, Mutex, MutexEntry, ReadyQueue and ConditionQueue. The MutexEntry extends the *net.jini.core.entry.Entry* interface, the superclass of all objects that can be stored in a JavaSpaces service, with two extra fields: a counter to be used in the Barrier to count the number of processes that arrive and a string value acting as a unique key identifying each distinct MutexEntry object in a JavaSpaces service.

Each instance of class Mutex contains a MutexEntry and provides mutually exclusive methods such as `lock()` and `unlock()` to access the MutexEntry instance. These two methods are implemented based on two blocking methods (`take()` and `write()`) in the JavaSpaces API. The `take()` and `write()` methods provide processes with synchronized access to entries stored in a JavaSpaces service. Each time an instance of class Mutex is created, a distinct MutexEntry instance is stored in the JavaSpaces service. Processes coordinated by the mutex invoke its `lock()` method to acquire the MutexEntry. One process will eventually succeed and remove the MutexEntry from the JavaSpaces service, while the others will be blocked until the MutexEntry is available again. The mutex lock can be released by the owner process (the MutexEntry is written back to the JavaSpaces service) so that others can compete for it.

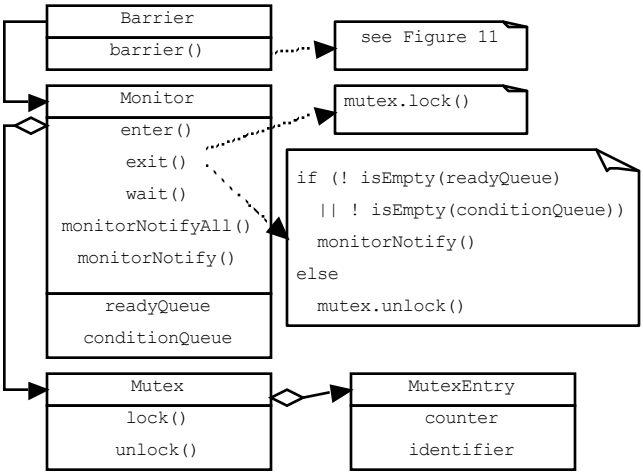

Figure 10. The CO_2P_3S distributed synchronization mechanisms

Our distributed monitor is based on our distributed mutex. A group of processes can wait on some conditions for coordination, analogously to Java threads using a thread monitor. A distributed monitor has one mutex and two queues that store blocked processes. In the Java thread model, the JVM associates one monitor for each object automatically, while in this case a distributed monitor has to be generated explicitly for each use.

Here is a behavioral description of our distributed monitor. Instead of using the `synchronized(Object)` syntax to guard a code block, we use `monitor.enter()` and `monitor.exit()` to signify the entry and exit of a code block guarded by the monitor. A monitor contains two queues to store waiting processes. A ready queue stores processes that are ready to continue their computations. Processes that are waiting for certain conditions to be satisfied are stored in a condition queue. If a process successfully obtains the mutex lock to enter into the code block, but later it finds that a certain condition is not yet satisfied, the process simply invokes `monitor.monitorWait()`, which places the process into the condition queue and releases the mutex lock. This is analogous to calling the `wait()` method in a standard Java program that uses threads. No queue is used to store the processes waiting for the mutex lock, as such a function is provided implicitly by the blocking mechanism of the `take()` method implementation of the JavaSpaces service.

If a set of processes are trying to acquire a mutex lock using `take()` and it is unavailable, all the processes will be blocked. When the mutex lock becomes available in the JavaSpaces service, one `take()` method will return and its process will be awakened. The specific order in which the processes are chosen depends on the implementation of the JavaSpaces service.

If a condition is true when one process enters into a code block, the process can invoke `monitor.monitorNotify()` or `monitor.monitorNotifyAll()` to wake up one or all of the waiting processes. Awakened processes are transferred from the

condition queue to the ready queue. Again, this is analogous to the Java threads approach.

Upon finishing the code block, a process must call `monitor.monitorExit()`. This causes the monitor to pick a process in the ready queue to be activated. If this queue is empty, the monitor will release the mutex lock to let processes waiting outside the code block compete for it.

Since the behaviors of a distributed monitor are so similar to the Java threads monitor, they are familiar and simple to use. Since pattern designers make extensive use of a Barrier class when designing the SM versions of their patterns, a distributed Barrier class has also been added to our distributed tool-chest. A distributed barrier uses a monitor to guard concurrent accesses to the barrier counter in the monitor's MutexEntry. Figure 11 is the key method in the distributed Barrier class. Figure 11 is also an excellent illustration of how similar the distributed programming syntax is to Java thread programming syntax.

```
public void barrier () {
   monitor.enter();
   counter = this.getBarrierCount() + 1;
   if (counter == procGroup.length) {
      monitor.monitorNotifyAll();
   }
   else {
      this.setBarrierCount(counter);
      monitor.monitorWait();
   }
   this.setBarrierCount(0);
   monitor.exit();
}
```

Figure 11. A distributed barrier

5. ENHANCING OUR INFRASTRUCTURE FOR PERFORMANCE

This section summarizes our work on enhancing RMI's performance by employing a more efficient and faster serialization routine. RMI was introduced into JDK1.2 to seamlessly incorporate a distributed object model. Combined with Java's dynamic class loading, object mobility, extensive security model, and platform-independence, RMI provides a convenient way of building Internet client/server applications. However, RMI does not offer high performance for applications on high-speed networks. There are two main obstacles to high performance: an inefficient transport subsystem and slow object serialization. Although using technologies such as the HotSpot adaptive compiler, JIT compiler, and Java native compiler can improve an RMI application's execution performance somewhat, the time spent doing object serialization and communication still occupies a significant amount of the total execution time of one RMI call.

RMI is unable to fully exploit network hardware resources, since it is implemented almost entirely in Java and TCP sockets, with only sparse use of the Java native interface calls to access low-level buffers. For example, RMI cannot gain performance by running on Myrinet or some other fast user-level networks.

JDK-serialization is a key component in RMI to implement the argument passing semantics of remote method invocations. However, JDK-serialization performs many computations that are redundant in the context of high-performance computing. For example, when an object is serialized, the object's class information is serialized as well.

Other researchers have worked on improving JDK-serialization. Good examples include UKA-serialization [31], the Manta project [21], and Jaguar [47]. The first two approaches apply a new wire format and require explicit serialization and de-serialization methods for each class. The third approach uses pre-serialized objects whose memory layout is already in a serialized form. Details of how our approach differs from these approaches can be found in [44]. However, in a nutshell, we wanted an approach that could be used with the existing RMI and that does not require the programmer to write any serialization code. In fact, our approach does require a slight modification to RMI, but we have transparently encapsulated the change. We provide a drop-in replacement for the JDK-serialization class (`ObjectInputStream`) and the de-serialization class (`ObjectOutputStream`). We use the Java command-line options to override the bootstrap classpath to insert our own classes into the existing bootstrapping class search path. Fortunately, all the execution commands are encapsulated in the scripts launched by the CO_2P_3S environment.

We perform four optimizations.

1. compact the class information,
2. remove security checks,
3. compact references to class information, and
4. take advantage of homogeneous arrays, where possible.

The idea of using compact class information has been applied in several other research projects. We record only the fully qualified name (such as java.util.Vector) for each different class, since all processors in a parallel application have a local copy of the same version of each class file.

In a high-performance computing environment, each processor used in an application has trust in the other processors so that objects can flow freely between them. Therefore, we removed all the security checks in our serialization and de-serialization processes.

Our third optimization is to compact shared references to classes. In an array of Integers, the standard serialized data stream includes many references that point to a serialized representation of the Integer class. JDK-serialization uses four bytes for a reference to this common class information. Thus, each element has a redundant 4-byte reference. Each array element uses four bytes to store an int value, so the common class references represent a storage overhead of almost 100%. We solve this problem by storing distinct object references in a hash table with indexes of size one or two bytes. Each entry has two attributes: one is a sequence number that reflects the order that the distinct reference was stored in the hash table. The other is the reference content. The first time a reference is encountered during serialization, it is written to the stream in full, headed by a short int with value -1. The new reference is also stored in the hash table with a sequence number. During serialization, if an identical reference is to be written, the sequence number of this reference is retrieved from the hash table and written into the stream. During de-serialization, if a –1 is encountered, the next 4 bytes are treated as the content of a reference. New references are appended to the end of a reference array in the order read. If a positive number (a sequence number) is read, de-serialization uses the entry in the

reference array indexed by this number. If the number of different references is less than 256, the sequence number can be a single byte. Thus the 4-byte reference is compressed to 1 byte. However, if the number exceeds 256, the hash table grows in size along with the sequence number; the resulting compression rate will be reduced from 4 to 2. In the rare case that the number of distinct classes whose objects must be serialized grows beyond 65,535, sequence numbers can grow to 3 bytes. The price of adding the hash table is one extra memory access to process each object reference during serialization and de-serialization. However the reduced transfer time outweighs this increased overhead in all of our experiments.

The fourth optimization that we used is to remove class references from individual array elements altogether, when we can prove before serialization starts that an array is homogeneous. Like most other object-oriented languages, Java supports type substitutability for arrays. Each element of an array with static type `classA[ ]` can be an instance of any subclass of `classA` at runtime. Therefore during the serialization process, the runtime type of each element must be computed and serialized even if all the elements end up having the same type. The dynamic type computation of array elements is expensive, especially when the array size is large. If the array elements are heterogeneous at runtime, such computation is necessary. However, if we know the array elements are homogeneous ahead of time, we can omit the type computation. In Java, a class marked as final can never be subclassed. Therefore, an array whose component class is declared as final must be homogeneous. We call such an array, a final array. We have changed the wire format of a serialized final array. A final array contains only the class of the array (which includes the element class), followed by the length, followed by the elements with no class references. In Java all the primitive wrapper classes like Integer and Float are final classes so in practice the opportunity for savings is large.

In the case where an array is not final, a reference for each element class (dynamic) must be stored. However, if this element is an object with fields (instance variables) that are primitive or instances of final classes, this approach is applied to these fields. We cache the element class information, which includes the class name and the fields information (final or not final). By applying the information to each array element, the serialization cost can be reduced by eliminating redundant class information and unnecessary tests.

One of the tests we did to evaluate our new serialization was to re-run the tests described in Section 4.2, that compared the speed of RMI with Java Sockets. The results are shown in Table 5. Although our custom serialization does not make RMI as fast as Java sockets, the advantages of RMI described in Section 4.2 are significant enough to outweigh the shrinking performance advantage of Java Sockets. Other performance results can be found in [44].

We also ran additional tests to compare our enhancements directly to UKA serialization work [31] and to standard JDK-serialization outside of the CO_2P_3S environment. Four kinds of data were used in the tests, an array with 1000 `TransportableTree` elements, one `TransportableTree` object, an array with 1000 `TestClass` elements and one `TestClass` object. The `TransportableTree` class implements the `uka.transport.Transportable` interface. The interface `uka.transport.Transportable` is the identifying interface for UKA-serialization to recognize uka-serializable objects. A `TransportableTree` has two `int` fields and two `TransportableTree` fields. This class also contains a set of methods that are generated automatically by a preprocessor. These methods are invoked by the UKA-serialization routine to reduce runtime type checking. These methods are transparent to JDK-serialization and CO_2P_3S-serialization. The level of the tree is chosen to be two, so that an instance of `TransportableTree` in the test program contains 7 nodes (the first level is 0).

Table 5. Comparison of standard and CO_2P_3S serialization (milliseconds)

	Objects	Strings	Integers
Standard serialization	36.5	4.6	6.3
CO_2P_3S serialization	31.4	4.2	5.6
% faster	16%	9%	13%
Java sockets	26.6	4.1	3.5

Class `TestClass` contains one `int` field and 5 `Integer` fields and only implements the `java.io.Serializable` interface. UKA-serialization will not recognize instances of this class and will just pass them to the standard JDK-serialization routine. Both the `TransportableTree` and `TestClass` class were declared as final in order to use the aggressive homogeneous array compression scheme of CO_2P_3S-serialization.

Table 6. JDK, CO_2P_3S, and UKA serialization (microseconds and bytes)

	Class	Serial	De-Serial	Total	Length
JDK	Tree [ ]	28460	35800	63260	106123
CO_2P_3S	Tree [ ]	26430	36110	62540	**71033**
UKA	Tree [ ]	25070	33030	**58100**	106084
JDK	Tree	440	290	730	188
CO_2P_3S	Tree	430	180	**610**	**112**
UKA	Tree	320	310	630	140
JDK	Test [ ]	22690	26570	49260	10061
CO_2P_3S	Test [ ]	19590	26330	**45920**	**5025**
UKA	Test [ ]	26890	26670	53560	11064
JDK	Test	360	400	760	38
CO_2P_3S	Test	350	180	**530**	**22**
UKA	Test	720	550	1270	41

For each serialization scheme, three times were recorded: the serialization time, the de-serialization time, and the sum of these two, as well as the length of the serialized data. The hardware and runtime configuration for running the experiments is the same as described in Section 4.2. Table 6

shows the results. The fastest time and shortest length is indicated in bold font for each separate test. Although CO_2P_3S-serialization is not the fastest in all cases, the total time is always faster than standard JDK-serialization and is very competitive with or beats UKA while being plug-compatible with standard JDK. It also provides the shortest serialized data in all cases.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a critical extension of the generative design pattern approach to parallel computing that supports the generation of distributed-memory code. This extension adds architecture-independence to the advantages of the GDP approach, while maintaining the generality and performance principles that are paramount in the GDP philosophy. We have described a new distributed-memory runtime environment for GDP that has been built from standard infrastructure components: Jini, RMI and JDK-serialization. We have introduced distributed synchronization primitives that mirror the familiar Java Monitor model that is used for thread programming. We have created distributed-memory versions of four GDPs: mesh, wavefront, phases and distributor that generate tunable code with reasonable parallel performance. We have introduced pluggable replacements for JDK-serialization that improve performance to the same degree as other non-pluggable approaches, like UKA-serialization.

There is more work that can be done to improve the performance of GDP generated code for distributed-memory architectures. For example, in our implementation, a JavaSpaces service stores all the shared data and acts as a medium for indirect message passing. This design is valid and provides reasonable performance for medium-scale parallel processing. However, if there are a large number of participants, concurrent accesses to shared data will result in a serious performance problem. There are two possible ways of solving this problem.

First, we could remove the JavaSpaces service and use explicit message passing for exchanging information and pass the mutex lock as a token. A tree algorithm or butterfly algorithm [48] can be used to reduce the number of messages exchanged. The Java Native Interface (JNI) [43] technology can also be used to improve communication performance.

Second, we could use a distributed-memory version of the JavaSpaces service [16]. Currently, there is one called GigaSpaces [18], which can act as a shared-memory layer for distributed systems and which has the same interface as a JavaSpaces service. Our implementation can be ported to GigaSpaces with only minor changes, yielding better performance for large-scale distributed computing.

Although performance improvements still need to be made, a new avenue for exploration is emerging for high-level models and tools. By abstracting out the architectural details using parallel design patterns, CO_2P_3S now supports both shared-memory and distributed-memory application development – a claim that few (if any) other parallel program development tools can make. We recommend this approach to other tool-builders. However, the usefulness of these tools is still limited by the number of parallel design patterns supported. We are actively working on enriching this set and we invite others to join us.

Can "ordinary programmers" create new generative design patterns? We think so. As anecdotal evidence about how difficult and time-consuming it is to write generative design patterns, an undergraduate summer student in our laboratory implemented five design patterns in three months. This included all of the time to read about and understand the descriptive (non-generative) versions of the design patterns, learn what generative design patterns are, figure out what parameters to define in each generative design pattern, learn to use CO_2P_3S and MetaCO_2P_3S, code the patterns and add them to CO_2P_3S using MetaCO_2P_3S. By the end of the process, the student was able to write a generative design pattern in about a week.

7. ACKNOWLEDGEMENTS

This research was funded in part by the National Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (ICORE).

8. REFERENCES

[1] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating parallel programs from the wavefront design pattern. In *Proceedings of 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, CD-ROM 1-8, 2002.

[2] H. Bal, F. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems, *IEEE Trans. on Software Engineering*, 18(3):190-205, 1992.

[3] K. Beck and R. Johnson. Patterns generate architecture. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pp. 139-149. Springer-Verlag, 1994.

[4] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998.

[5] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18-32, 1998.

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

[7] S. Bromling, S. MacDonald, J. Anvik, J. Schaefer, D. Szafron, K. Tan, Pattern-based parallel programming, *Proceedings of the International Conference on Parallel Programming (ICPP'2002)*, August 2002, Vancouver Canada, pp. 257-265.

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

[9] N. Carriero and D. Gelernter. Linda in context. *Commun. of the ACM*, 32(4):444-458, October 1989.

[10] X. Chen and V. H. Allan. MultiJav: a distributed shared memory system based on multiple Java virtual machines. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pp. 91-98, 1998.

[11] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT Press, 1989.

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

[13] A. Eden, Y. Hirshfeld, and A. Yehudai. Towards a mathematical foundation for design patterns. Technical

Report Technical Report 1999-004, Dept. of Information Technology, University of Uppsala, 1999.

[14] R. Finkel and U. Manber, DIB - A Distributed Implementation of Backtracking. ACM TOPLAS, April 1987, pp. 235-256.

[15] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 472-495. Springer-Verlag, 1997.

[16] E. Freeman and S. Hupfer, Make room for JavaSpaces, Part 1: Ease the development of distributed apps with JavaSpaces, http://www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology.html

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

[18] GigaSpaces Technologies. GigaSpaces cluster white paper, http://www.gigaspaces.com/download/GSClusterWhitePaper.pdf, 2002.

[19] A. Grimshaw. Easy to use object-oriented parallel programming with Mentat, *IEEE Computer*, pp. 39-51, May, 1993.

[20] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22-35, 1988.

[21] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming, *Programming Languages and Systems*, 23(6):747-775, 2001.

[22] S. MacDonald. From patterns to frameworks to parallel programs. Ph.D. thesis, Dept. of Computing Science, University of Alberta, 2002.

[23] S. MacDonald, J. Anvik, S. Bromling, D. Szafron, J. Schaeffer and K. Tan. From patterns to frameworks to parallel programs, *Parallel Computing*, 28(12);1663-1683, 2002.

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

[25] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns, *17th IEEE International Conference on Automated Software Engineering (ASE)*, pp. 23-34, 2002.

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

[27] M. Massingill, T. Mattson, and B. Sanders. A pattern language for parallel application programs. Technical Report CISE TR 99-022, University of Florida, 1999.

[28] ModelMaker Tools. Design patterns in ModelMaker. http://www.modelmakertools.com/mm_design_patterns.htm.

[29] J. E. Moreira, S.P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, Java programming for high-performance numerical computing. *IBM Systems Journal* 39, 2000, pp. 21-56.

[30] J. Newmarch. *A Programmer's Guide to Jini Technology*. Apress, November 2000.

[31] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495-518, May 2000.

[32] J. Reynders, et al. POOMA: A framework for scientific simulations of parallel architectures, *Parallel Programming in C++*, G. Wilson and P. Lu (editors), pp. 547-588, MIT Press, 1996.

[33] 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.

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

[35] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361-372, 1992.

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

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

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

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

[40] F. Sommers. Activatable Jini Services, Part 1: Implement RMI Activation, http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-jinirmi.html

[41] Sun Microsystems. Java Remote Method Invocation Specification, JDK 1.1, http://java.sun.com/products/jdk/rmi_ed, 1997.

[42] Sun Microsystems. Jini Architectural Overview, 2001. http://wwws.sun.com/software/jini/whitepapers/architecture.pdf.

[43] Sun Microsystems. JNI Specification, 2000. http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html

[44] K. Tan, Pattern-based parallel programming in a distributed memory environment. Master's thesis, Dept. of Computing Science, University of Alberta, 2003.

[45] TogetherSoft Corporation. TogetherSoft ControlCenter tutorials: Using design patterns. http://www.togethersoft.com/services/tutorials/index.jsp.

[46] B. Venners. *Inside the Java 2 Virtual Machine*. McGraw Hill, 2nd edition, 1999.

[47] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519-538, 2000.

[48] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1st edition, 1999.