Pattern–based Object–Oriented Parallel Programming

Steve MacDonald*

1 Introduction and Motivation

Parallel programming offers substantial performance gains to those willing to take up the challenge. By properly using additional processors, either in a single multiprocessor machine or in a network of workstations, we can execute programs faster than they would execute sequentially. We can apply this benefit to either reduce the execution time of a large program or run larger, more detailed problems in the same amount of time.

Unfortunately, this benefit comes at a steep price. Parallel programming is a difficult task, combining the complexities of sequential programming with additional issues. The programmer must now create and coordinate the processes that are participating in the computation. Parallel activities must be properly synchronized to ensure correct results. The user will also have to deal with any nondeterminism introduced by the concurrent activities, since the program may no longer execute in a consistent order each time it is run. This last problem also hampers debugging since a program may only fail on specific orderings that can be difficult to reproduce.

To combat this complexity, we need a set of tools that aid in the development of parallel programs. One such tool is a *parallel programming system* (PPS). A PPS can be responsible for managing some of the added complexity by providing a simple parallel programming model to the user that eases or removes some of the above issues. It may further help the user by providing a complete toolset for debugging, executing, and tuning programs. This proposal discusses advances in the state–of–the–art in parallel programming systems.

Recently, state–of–the–art PPSs have started to concentrate on *template–based* (or, as they are now called, *pattern–based* [16]) parallel programming systems. By observing the progression of these systems, we can clearly see the evolution of pattern–based computing technology. One of the first attempts, FrameWorks [39], allowed users to graphically specify the parallel structure of a procedural program in much the same way they would solve a puzzle, by piecing together different components. The programming model of FrameWorks was low–level and placed the burden of correctness on the user. This research led to Enterprise [35], a PPS that provided a limited number of templates that could be composed in a structured way. The programming model in Enterprise was at a much higher level, with many of the low–level details handled by a combination

*Sections of this paper will be published in the Proceedings of the 1997 International Scientific Computing in Object–Oriented Parallel Environments Conference (ISCOPE'97) [24].

of compiler and run–time technology.

In this progression, we can see more emphasis placed on the *usability* of the tools rather than raw performance gains [44]. Each successive system reduced the probability of introducing programmer errors. However, since performance considerations cannot be ignored, each successive system also supported incremental application tuning. A related performance issue is *openness* [40], where a user is able to access low–level features in the PPS and use them as necessary. These concerns led to a critical evaluation of pattern–based systems that provides the motivation for the new system, CO_2P_3S[1].

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

This paper is organized as follows. Section 2 is an overview of other parallel programming systems. Section 3 presents the architecture for incremental tuning. Section 4 presents the CO_2P_3S model by using an example application. Section 5 evaluates the CO_2P_3S system and the PPSs of Section 2 using established criteria for pattern–based parallel programming systems. Section 6 is an overview of other related work in the design pattern and parallel programming fields and compares this work to the new project. Section 7 proposes thesis research within the project. Some conclusions are presented in Section 8.

2 Parallel Programming Systems

This section provides an overview of the literature on parallel programming systems. This review emphasizes object–oriented tools and systems since they are the main focus of the research. However, procedural systems are also mentioned when they are applicable.

We also describe some of the limitations of these systems. A more thorough description of the problems is

[1] Correct Object–Oriented Pattern–based Parallel Programming System, pronounced “cops”.

delayed until Section 5.3, where we compare the systems using a set of characteristics of good pattern–based parallel programming systems.

2.1 Pattern–based Parallel Programming Systems

This section surveys the relevant literature on pattern–based parallel programming systems. This is not intended to be a complete survey of all existing systems, but rather a cross–section of the major research in the area.

FrameWorks FrameWorks [39] is a template–based distributed programming system based on a set of *modules* communicating via remote procedure calls. A module is a set of procedures, including a special *entry* procedure, written in a superset of a high level language (in this case, C). Variables cannot be shared between modules. Each FrameWorks program must also include a *main* module that contains the `main()` procedure.

The specification for a module also includes three template components: an *input* template, an *output* template, and a *body* template. These three templates are used to create appropriate communication, scheduling, and synchronization code.

The input and output templates define how a module interacts with the other modules in a FrameWorks application, specifically how a module receives its input and sends its output to the other modules. Three input modules were defined:

1. an *initial* template that does not receive input from any other modules,
2. an *in–pipeline* template that receives input from any of its input modules and services the requests first–come–first–served, and
3. an *assimilator* template that requires input from each of its input modules before it invokes its entry procedure.

There are three output templates as well:

1. an *out–pipeline* template that can send its output to one of the (possibly many) output modules,
2. a *manager* template that manages a fixed number of identical output modules, and
3. a *terminal* template that does not send output to any other modules.

Body templates are optional templates that define the internal behaviour of a module. FrameWorks defined two body templates:

1. an *executive* template that instructs the module to direct its input, output, and error streams to the user's terminal, and

2. a *contractor* template that dynamically seeks out and uses idle workstations (called *employees*) for compute–intensive applications.

Modules communicate with other modules by explicitly invoking them with a new `call` statement. Invocations that do not return values are asynchronous. Otherwise, the calling modules suspends waiting for results, which must be returned using a `reply` statement in the called module. The data exchanged by different modules is specified as a *frame*, which resembles a C structure except that it cannot contain pointers. A module call uses one frame for the input parameters and one frame for any return values.

FrameWorks suffered from several limitations [40]. The system was not open since users cannot access the run–time facilities. Foremost, though, FrameWorks exhibits several usability–related problems. There were some non–intuitive constraints on the composition of the three templates for a module. The user's code and the templates were tightly coupled; any changes in one had to be reflected in the other. Finally, the user was responsible for correctly packing data into the frames used for communication between modules.

Enterprise Enterprise, the successor project to FrameWorks, is a parallel program development environment with support for design, coding, compiling, executing, debugging, analyzing, and replaying parallel programs [35, 20].

The programming model of Enterprise uses asynchronous procedure invocations synchronized with *futures* [19]. We can demonstrate this model with an example code segment:

```
result = f(param1, param2, ..., paramN) ;
/* Other code */
a = result + b ;
```

In the sequential execution of this code, the caller invokes the function `f()` with the supplied arguments and waits for the function to return before resuming. In the Enterprise model, if the function `f()` was annotated as a parallel call, the caller marshals the arguments into a message and sends them to the process responsible for executing `f()`. The results are assigned futures for later synchronization. The caller then continues to execute, running its own code concurrently with the parallel call. This continues until the caller tries to use the value of the future, forcing it to *resolve* itself. Now, if the parallel call has not completed, the caller blocks waiting for the results. If the results are already available, the caller continues executing. Since the generation and resolution of futures is implemented by the Enterprise compiler, the user writes (almost) normal C code.

The futures model in Enterprise also takes into account parameters passed by reference (done through pointers in C). However, to pass pointers, the user must also specify the number of elements to be passed and can optionally specify a direction to optimize the size of the messages between processes. This pointer data cannot contain further pointers.

Enterprise also has a *meta–programming* model that allows a user to specify the parallel structure of a program using a series of composable templates called *assets*. The meta–programming model is based on an analogy between parallel programming structures and the structures found in a business organization. Enterprise provides the following assets:

Enterprise: This asset represents a complete program. Initially, the entire program consists of a single *individual* asset. Only a single copy of this asset exists in a program.

Individual: This asset represents a worker in a parallel program. An individual contains no other assets. An individual has source code and the name of a procedure it is responsible for executing. The procedure is executed to completion when invoked. An individual can be *replicated* to provide multiple copies of the same asset. Further, an individual can be *coerced* to another asset type, replacing the individual with another asset. An individual can be coerced into a *line*, *department*, or *division*.

Line: This asset represents an assembly line. It contains a set of heterogeneous assets in a specified order. Note that these assets need not be individuals. Each asset in the line can call the next asset, usually passing it some data to be refined at each stage of the line. The first asset in a line is a *receptionist*, which is similar to an individual in that it executes its procedure to completion. Subsequent invocations of the pipeline only wait for the receptionist to finish executing, not the entire line. A line can be replicated.

Department: This asset provides a master/slave template. The department consists of a receptionist and a set of heterogeneous assets. Again, the assets need not be individuals. The receptionist is invoked and can in turn invoke any of the assets in the department. Subsequent invocations of the department must wait for the receptionist to finish executing. A department can be replicated.

Division: This asset provides a recursive divide–and–conquer template. The division consists of a receptionist and a *representative*, which is similar to an individual and represents a leaf node in the recursion. Each asset in the division executes identical code. At non–leaf nodes, a recursive call is translated into a remote call. At leaf nodes, all recursion is performed locally. The breadth and depth of a division can be modified through a combination of coercion and replication. A representative can be coerced into a division, increasing the depth of the division. The divisions and representatives can also be replicated to increase the breadth.

Service: This asset represents any resource that must be shared among assets. It does not contain any assets, but any asset in a program can call it. A wall clock is a good example of this kind of service.

It is important to note that the meta–programming model can only build acyclic graphs where the flow of control is always down. This ensures that the communication implied by the asset graph cannot result in a deadlock.

Once the user has specified the asset graph and the code for the application, the Enterprise precompiler transforms the sequential code into a parallel program. The asset graph is used to determine the procedure calls that must be transformed into remote calls. The invocation of the call is replaced with a call to another compiler–generated routine that marshals the arguments and sends a message to the remote process that will execute that procedure. The compiler also detects the futures resulting from the call and ensures that they will be resolved before their values are used. The actual communication and synchronization is performed in the run–time system.

Enterprise improved on FrameWorks in several ways [40]. First, an asset specifies the template completely, rather than in three parts as required by FrameWorks. Enterprise provided the user with more templates. The programming model was familiar, based on (almost) sequential C code. Enterprise decoupled the templates from the user code, allowing each to evolve more independently. Finally, there are more correctness guarantees for Enterprise programs. The system handles the packing and unpacking of parameters, including some support for pointer data. Enterprise also guarantees that a program matches its meta–program at compile time and that the meta–program is deadlock–free.

However, Enterprise still has its limitations. The system is not open, as the run–time system was designed as a compiler target and provides a minimal interface. The most serious limitation is that users must rewrite their code to take full advantage of the futures–based programming model. Any accesses to a future must be separated from the creation of that future. Achieving this separation may mean rewriting some basic sequential code, such as loops summing the results of a series of parallel calls.

Tracs In contrast to the FrameWorks and Enterprise systems, the Tracs system [3] is aimed at more sophisticated users. Programs are represented as directed graphs where the nodes represent computational units called *tasks* and edges represent communication. Each task has input and output ports, where each port has a message type associated with it. Tasks typically communicate by either sending messages across uni–directional channels or by using a remote procedure call mechanism to invoke services at another task.

The user creates a program by specifying up to three components: *message models*, *task models*, and *architecture models*.

The message models define the message types for ports on a task. A message model is similar to a frame in FrameWorks; each model is a structure containing all the necessary parameters. Unlike FrameWorks, though, the message models are automatically mapped onto the XDR data definition format, which allows pointer data and complex data types to be used. The user does not need to define new message models for each application; they can be saved to disk and recalled for later applications.

The task models define the nodes in the program graphs. Each task model defines a set of input ports, output ports, and services. Each service and port is associated with a message model and local name. The input and output ports define the static structure of the task; one is needed for every external connection. A task may define services that are used to export specific functionality to any number of other tasks. A task model also includes code to be executed by the task, which uses the local names to send message models through the supplied ports or invoke a service on another task. Like message models, task models can be kept and used in other applications.

Finally, the user specifies the architecture model. This model specifies the parallel structure of a program independently of the actual tasks, ports, and services. These models represent the templates or patterns in the Tracs system. An architecture model consists of a program graph with formal message and task models, akin to formal parameters in a procedure. The formal models define names for each component but do not specify an implementation. The user fills in the parameters for the architecture model, supplying the names of the actual message and task models. However, the graphs used in the architecture model are static; the supplied patterns cannot be generalized and instantiated with some specified parameters. For example, a user cannot create an architecture model that creates an `n`–stage pipeline where `n` is specified when the model is used. The user interface does compensate for this shortcoming by providing operations that allow repetitive structures to be easily built. Again, like the previous models, architecture models can be loaded from other applications or from models supplied by Tracs.

The programming model in Tracs is explicit message passing or service invocation. Messages are sent to a port rather than another task, where the port is referenced using the local name supplied in the task model. These messages are uni–directional and can be either synchronous or asynchronous.

The Tracs system provides a feature that allows a user to use sequential code. This feature may only be used with task models that define any number of input ports and a single output port. The user defines a procedure that has one argument for every input port. When data arrives on each input port, the procedure is invoked with the arguments set using the data from the ports. The return value of the procedure is sent to the output port. This feature is implemented by using a wrapper procedure that reads data from each input port, invokes the user–supplied sequential function, and sends the results to the output port. By modifying this wrapper code, it is also possible to define other behaviours, such as a master/slave, pipeline element, and grid element.

Finally, the Tracs system provides support for compiling, debugging, and replaying programs.

The principle failing of Tracs is the intrusive programming model, which is a departure from the call/return sequential model. This problem appears in both the explicit message–passing mode and sequential code feature. Using the sequential code feature, the return value of a task does not return to the caller but is a mechanism for sending data to another task. Also, the code does not invoke other tasks. The use of the message models for parameters is intrusive. Finally, Tracs also fails to make correctness guarantees unless the behaviours in the sequential code option are used.

Parsec The Parsec system [13] is a pattern–based system similar to Cole's algorithmic skeletons [11] and the pragmatic implementation machines of PIE [33]. The system provides virtual machines (called skeleton–template–module objects or STMs) providing an abstract computer architecture that supports different parallel programming templates, such as master/slave and task farms[2]. The virtual machines

[2]These virtual machines are similar to the virtual machines for Smalltalk and Java in that they provide an abstraction of the hardware and software architecture of the target machine. However, unlike Smalltalk and Java, they do not interpret and execute bytecode.

provide skeleton code for the supported template, which the user completes to implement an application. The virtual machines also require a set of parameters, some from the user and some derived from the topology of the underlying hardware, to determine the full implementation of the template and its mapping to the parallel architecture.

The parameters in Parsec are unique in that they are not all static values input from the user or derived from the hardware, although these kinds of parameters are the most common and easiest to handle. Parsec allows parameters to be the output of a specified external program. This program can be used to determine dynamic performance tuning parameters of a system that may be different for each execution of the application.

Parsec also generalizes the structure of its patterns by allowing the user to supply parameters to specify its shape and size. This features helps them achieve their goal of providing users with small development versions of their programs that can be easily scaled up to much larger production versions.

In terms of the programming model of Parsec, the patterns represent programs as graphs. Each node in the graph represents a *process*, which has associated *ports* which are bound to other ports via *channels*. The processes may be collected into *modules*, which may be hierarchically defined by subdividing a module into *groups* of logically related modules. The processes communicate explicitly using typed messages sent to ports. These message types are used to guide the marshaling of messages, which is done invisibly using XDR.

Parsec fails to provide an open system, encapsulating the template in the virtual machine and keeping it from the user. Also, the explicit communication model and message types do not map well to a sequential programming model.

DPnDP DPnDP (Design Patterns and Distributed Processes) [42] represents programs as a combination of design patterns and *singleton* nodes in a directed graph. Each node represents a computational module that contains message handlers that receive messages and invoke the appropriate module code. A module may be implemented using a design pattern. Furthermore, any singletons within the design pattern may be design patterns themselves. These nodes communicate explicitly through a set of ports.

The design patterns in DPnDP handle pattern–specific creation, communication, and synchronization, but application–specific communication is left to the user. For instance, in a task farm, the pattern code handles the distribution of work to the workers and the necessary synchronization, but the user must still send work and receive results in the master and slave code. Communication is performed through low–level interprocess communication primitives, which are mapped to a message–passing library such as PVM or MPI.

Patterns in DPnDP only specify the generic structure, such as an `n`–stage pipeline. The user instantiates the pattern with a set of parameters indicating the correct size and structure. Unlike Parsec, though, these parameters are static values input by the user.

The design patterns in DPnDP are used to generate code skeletons that the user augments with application–specific functionality by defining the message handlers. The skeleton code can be modified by the user, promoting openness. Openness is further enhanced by allowing the user to access the underlying communications library, which allows a program to use specific features of the library or allow outside programs to interact with DPnDP applications.

One important aspect of the design patterns is that they cannot be distinguished from singleton modules. This feature allows any singleton to be replaced with a design pattern. DPnDP refers to this replacement as *composition*, which allows a program to be specified as a graph of singletons and patterns. It is also possible to *refine* patterns, modifying their behaviour while maintaining the overall structure and behaviour of the pattern. An example of refinement is replicating a node in a pipeline.

Another critical aspect of the design patterns in DPnDP is that they all present and implement identical interfaces. This uniformity forces the patterns to be context–insensitive.

This uniformity also leads to an important contribution of DPnDP. Given the common interface, it is possible for a user to create new design patterns and add them to the existing set. DPnDP provides a set of C++ classes that users can derive from to specify new patterns [41]. The user supplies methods to create the structural and behavioural parameters for the new pattern, specifying the kinds of nodes used in the pattern, initializing all the design patterns used in the new pattern, and a final method to generate the interconnection between the modules in the pattern based on the structural parameters. These four methods, combined with the remainder of the design pattern implementation classes, provide all the code generation and implementation aspects necessary to introduce new patterns into the pattern library. The only missing aspect is that the user interface does not recognize these new patterns, which can be corrected by placing this information in an external database.

DPnDP has two main limitations. First, the programmer marshals the data sent between modules, leaving the correctness of this important code to the user. Second, the explicit communication model is intrusive. This intrusiveness appears in the recursive divide–and–conquer pattern defined by DPnDP. To correctly use this pattern, the user must specify a different behaviour for each kind of node in the recursion: the root node, interior but non–root nodes, and leaves. This is counter–intuitive from a sequential programming perspective, where we would expect each node to execute the same code. However, each of the three kinds of nodes have different communication requirements. The root receives input from an external module and sends messages to its children. Interior nodes receive input from another interior node and sends messages to its children, and the leaves receive input from another interior node but generate no further messages. The explicit communication model makes it necessary to make this distinction.

2.2 Non Pattern–based Parallel Programming Systems

In addition to the pattern–based parallel programming systems from the previous section, there are many systems that provide a more visual but less structured approach to building parallel programs. Examples of this type of system are CODE [28], HeNCE [4], and VPE [29]. These systems typically represent programs as directed graphs, with nodes representing computation and edges representing interaction or communication. HeNCE graphs show control dependencies, data dependencies, and name scoping. A HeNCE node can execute only when all preceding nodes have finished. Each node can only receive input when it starts executing and send output when it finishes. CODE graphs show a modified dataflow relationship. A user can specify firing rules for a node which may only require data on a subset of the incoming arcs[3]. Like HeNCE, a node cannot communicate during its execution. VPE graphs are mainly intended for helping programmers visualize the structure of their applications rather than strictly controlling program execution. VPE nodes explicitly communicate through asynchronous messages using ports defined in the graphical interface, eliminating the need for a node in the program graph for each message.

These tools provide an easy way for the programmer to specify the parallel structure of a program. This guides the user when building the program. Also, it is possible for the run–time system of the tool to use the graph to help with communication by type–checking messages. Most of these systems also provide graphical user interfaces to help with the development of programs, providing the opportunity to include support tools. Lastly, the visual nature of the programming model lends itself to graphical performance analysis tools such as animation.

The main problem with these tools compared to their pattern–based counterparts is a lack of correctness with respect to the proper use of the program graphs. This results from the lack of structure in the graphs used to represent the structure of the program; the graphs can indicate the intended communication in an application but do not verify that it is used correctly. We can see this in the Tracs system, which allows the user to build patterns but makes no guarantees relating to their use. Another problem with these systems is that most define an intrusive programming model that does not follow the familiar sequential model.

However, it is possible to use the properties of a visual language to verify properties of resulting applications. Phred [5] describes the parallel structure of an application (the grammar of the Phred visual language) and accesses to shared repositories of data. The grammar and semantics of the visual model make it possible to determine if a program is deterministic by showing that the tasks in the system do not have conflicting accesses to a repository. However, this analysis was developed specifically for the Phred grammar. It is not clear if this can be extended to more general task graphs.

2.3 Summary

In all of the systems described in the previous two sections, three issues were consistently raised: program correctness, the intrusiveness of the programming model, and system openness. These areas are crucial to creating a usable parallel programming system. If a PPS can make some correctness guarantees, it relieves the programmer of some of the burden of writing parallel programs. Further, the more the programming

[3] In traditional dataflow systems, a node may execute only when data is available on all incoming arcs.

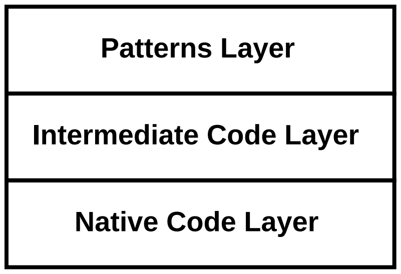


Figure 1: The architecture of CO_2P_3S.

model of a PPS strays from the sequential model, the more difficult it will be for users to learn how to use the system effectively. Finally, an open system allows users more opportunity to modify and tune their programs for additional performance. Unfortunately, the systems presented in this section generally target experienced users. These systems may prove to be daunting to new parallel programmers. However, systems that target new programmers typically do not provide either the control or the performance levels required by more advanced users.

With the CO_2P_3S system, we hope to improve the state–of–the–art in parallel programming system research by reducing these limitations. This work can ease the transition into parallel programming for novice users. However, we define an open architecture with multiple abstractions to address the usability and performance concerns of intermediate and advanced parallel programmers. The result will be a novel system that can be used by a broad range of users.

3 The Architecture of CO_2P_3S

The architecture of CO_2P_3S is shown in Figure 1. It consists of three layers: `Patterns`, `Intermediate Code`, and `Native Code`. These layers represent different levels of abstraction, where the abstraction of each layer is implemented by the one underneath.

Each layer is transformed during compilation to the layer underneath. At the pattern layer, the developer selects a pattern using a graphical tool. The PPS then generates a template for the parallel application and the user is restricted to providing sequential application–specific code at particular locations in the generated template[4]. At the pattern level, the PPS guarantees the correctness of the generated program by using conservative synchronization mechanisms that may not yield peak parallel performance.

Unlike other systems, CO_2P_3S provides programmer access to the second layer so that the templates themselves can be edited. From this layer down, the programmer is responsible for the correctness of the resulting program. To make this task simpler, we provide a high–level parallel programming language that is an extension of existing object–oriented languages such as Java and C++. This superset includes extra

[4]For the remainder of this paper, a template refers to generated code.

keywords to annotate parallel classes, specify concurrent activities and express synchronization requirements (using constructs such as asynchronous method invocation, threads, and futures). These keywords are specific to the current pattern.

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

Our architecture addresses the problems of non–intrusiveness, openness, and correctness. The system is non–intrusive because the user writes sequential code to fill out the templates. The pattern layer addresses correctness even more rigourously than other systems by generating template code from the design patterns. Since, at the pattern layer, the generated code cannot be modified by the user, we can strictly enforce the structure of the program. Combined with a run–time library, this code can offer a high–level model that frees the user from the low–level details of parallel programming and guarantees the correctness of the parallel structures.

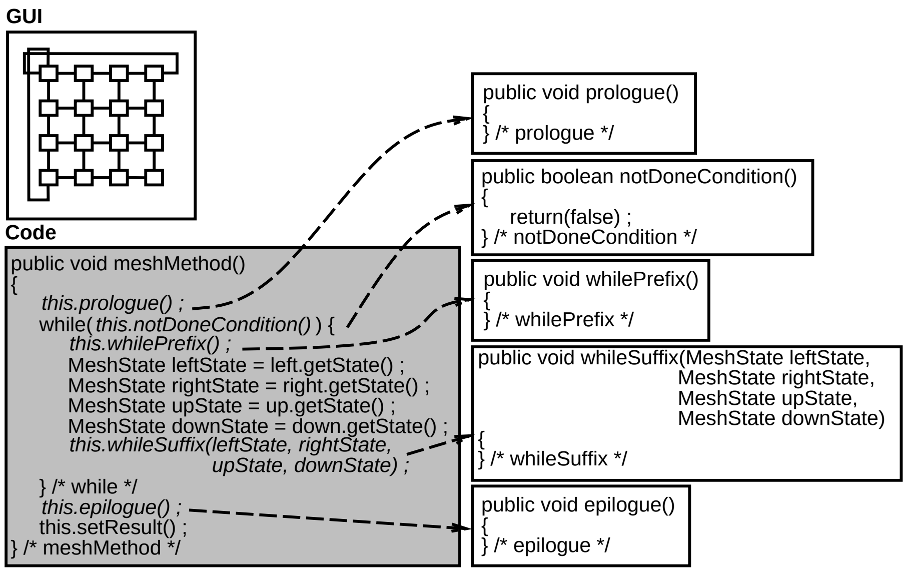
The subsequent layers of the system are intended to address the problem of openness by providing successively more access to both the generated code and the run–time system. The intermediate layer provides more control over the user program by providing access to the generated code of the first layer,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

The novelty in this approach lies in providing different layers of abstraction for incremental tuning. In contrast, most existing systems provide either no incremental refinement or simply give the user access to the underlying run–time system. We believe that providing intermediate levels of abstraction can provide several benefits. First, by generating correct template code at the first level, we can ensure that a user has a working parallel program before the tuning process begins. Second, it eases the tuning process by introducing the run–time system in smaller increments. These smaller increments provide better opportunities for novice or intermediate users to find a comfortable level of abstraction while still providing full access to the run–time system for experienced users. Lastly, it should always be possible to improve the performance of a program by using the abstraction of a lower level. If so, then the performance of an application should be more directly commensurate with programmer effort.

4 The Model

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

The first layer specification is shown in Figure 2. This layer consists of both a graphical representation of the pattern and the generated template code. The pattern itself has several parameters that must be given.



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

```
public void prologue()
{
} /* prologue */
```

```
public boolean notDoneCondition()
{
    return(false) ;
} /* notDoneCondition */
```

```
public void whilePrefix()
{
} /* whilePrefix */
```

```
public void whileSuffix(MeshState leftState,
                        MeshState rightState,
                        MeshState upState,
                        MeshState downState)
{
} /* whileSuffix */
```

```
public void epilogue()
{
} /* epilogue */
```

Figure 2: The first layer of CO_2P_3S. Shaded code is generated by the system and cannot be modified. Italicized methods are null and must be implemented by the user. Skeleton implementations are shown on the right.

The parameters include the width and height of the mesh and its boundary conditions. The boundary conditions are the most interesting of these parameters since they can affect the user code. The reason for this can be seen in the accompanying code fragment. It shows the main loop in the evaluation of the mesh, with the shaded part representing the generated code and italicized code representing hook methods that the user must implement. In the hooks, the code uses all four mesh neighbours. However, if the mesh is not fully toroidal[5], then not all the neighbours will exist and the supplied code is not applicable. To correct this, we allow the user to specify different code for the different special cases. In this paper, for simplicity, we will assume the mesh is fully toroidal.

One critical feature of the first layer is that the generated code encapsulates the communication and only allows the user to operate on the data. This feature allows our PPS to make certain correctness guarantees, such as ensuring that each element of the mesh is referenced. The code in Figure 2 demonstrates this property by getting the state from the mesh neighbours and communicating the results of the computation to a *collector* object responsible for collecting the results (encapsulated within `setResult()`). Another critical aspect of this code is that we provide sufficient hooks for users to completely implement their programs. This requirement must be evaluated on a pattern–by–pattern basis. For the portion of the mesh in Figure 2, we provide five hooks. `prologue()` and `epilogue()` are executed before and after the mesh computation and can be used for any initialization and cleanup, such as implementing instrumentation. `whilePrefix()` and `whileSuffix()` are executed for every iteration of the mesh algorithm and can be used to preprocess

[5]A mesh is fully toroidal when all neighbours of all elements are defined, usually by linking the first element of a row to the last element of that row and similarly for columns.

Code

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

public void whileSuffix(MeshState leftState, MeshState rightState,
                        MeshState upState, MeshState downState)
{
    // Take the average myself and all four neighbours and gather
    // timing information from a timer started in whilePrefix().
    this.value = (this.value + leftState.getValue() +
                  rightState.getValue() + upState.getValue() +
                  downState.getValue()) / 5.0 ;
    this.stopTimer() ;
    this.gatherStatistics() ;
} /* whileSuffix */
```

Figure 3: The second layer of CO_2P_3S. The italicized code is synchronization inserted by the transformation between layers.

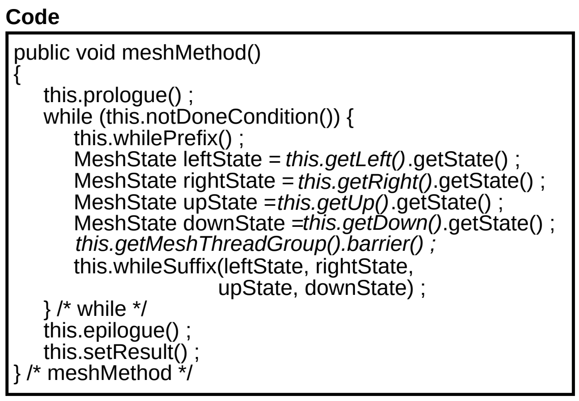
local data and perform the desired mesh operation. Finally, `notDoneCondition()` lets the user specify the terminating condition for an element of the mesh. The mesh stops executing when all the elements are finished. The default implementation of these methods is given on the right hand side of Figure 2.

Since the user hooks are given as methods that the user implements, it is possible to view the generated code as a *template class* like those in C++. The parameters for this template class are the hook methods.

From the first layer code, we generate the intermediate code of the second layer shown in Figure 3. The figure also contains an example of a method operating on data in the mesh, which would have been added by the user at the first level. For the mesh, the keywords `left`, `right`, `up`, `down` are defined to refer to the neighbours. The keyword `barrier` is also required to define the necessary synchronization for the mesh and is automatically inserted in the transformation of the mesh code from the first to the second layer. Other, more general keywords are also supported by CO_2P_3S.

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

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



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

Figure 4: The third layer of CO_2P_3S. The italicized code represents replaced keywords. The code for other methods is accessible but not shown.

than examining only the immediate neighbours.

Finally, from the second layer code of Figure 3, we generate the native code given in Figure 4. Native code replaces the keywords introduced by the second layer with library calls implemented by our parallel library. This replacement may be as simple as replacing the keyword with an accessor, as with the references to the neighbours of the mesh. Others may be a little more complex, such as replacing the `barrier` keyword with a call to the thread group holding the threads executing the mesh. Users are also free to use any library call or language feature available to them.

5 Evaluating the Model

In this section, we examine the 13 desirable characteristics of pattern–based PPSs described by Singh *et al.* and apply them to CO_2P_3S. Not all of the characteristics can be addressed at this time, as some require separate usability and performance studies that cannot be undertaken until our tool is fully implemented and mature enough for the user community. We first outline the characteristics and then evaluate CO_2P_3S with respect to them. Finally, we evaluate the systems described in Section 2.1 using the same criteria.

5.1 Basis for Evaluation

This section defines the characteristics of a good pattern–based parallel programming system. Like Singh *et al.*, we break the characteristics into three categories. We also use their short names for the characteristics, which are given in parentheses.

5.1.1 Structuring the Parallelism

These characteristics examine how users can structure the parallelism in their applications. There should be as few restrictions as possible. The ideal characteristics are:

1. Separation of Specification (Separation): It should be possible to specify the parallelism (design pattern) and application code separately, allowing each part of the specification to evolve independently.

2. Hierarchical Resolution of Parallelism (Hierarchy): This characteristic means that a design pattern can be refined by embedding other patterns within it.

3. Mutually Independent Patterns (Independence): Each pattern should be context insensitive so that it can be used in conjunction with any other pattern.

4. Extendible Repertoire of Patterns (Extendible): The user should be able to create additional design patterns. These user–defined patterns should not be discernible from system–defined patterns.

5. Large Collection of Useful Patterns (Utility): The system should have a large number of patterns that encompass a broad range of application areas.

6. Open Systems (Open): The user should be able to access low–level features of the programming system.

5.1.2 Programming

These characteristics are used to evaluate the style and structure of the application code written by the user.

1. Program Correctness (Correctness): The PPS should be able to make correctness guarantees about the user program. Common examples of such guarantees are absence of deadlock and deterministic execution.

2. Programming Language (Language): The system should use an existing, familiar programming language.

3. Language Non–Intrusiveness (Non–Intrusiveness): The user should not have to program around limitations in the parallel programming model. For instance, a user should not have to avoid pointers to accommodate a parallel programming system that uses a message–passing library in its implementation.

5.1.3 User Satisfaction

These characteristics focus on a combination of performance and usability concerns.

1. Execution Performance (Performance): The system should provide the best possible performance, within the bounds of the provided patterns.

2. Support Tools (Support): The system should provide a complete set of support tools, including design, coding, debugging, testing, and monitoring tools. Each of these tools should present a consistent model to the user.

3. Tool Usability (Usability): The programming system should be easy to learn and use.

4. Application Portability (Portability): Users should be able to port applications to several target architectures. This characteristic does not imply that performance will be equal on each architecture.

5.2 Evaluating CO_2P_3S

5.2.1 Structuring the Parallelism

CO_2P_3S addresses separation by expressing the parallelism diagrammatically and allowing application–specific code to be inserted into the pattern. In our case, the diagram is some form of *collaboration diagram*. Further, the CO_2P_3S approach of generating code that invokes user hooks provides more opportunity to re–use the hooks when the user changes the pattern. Since the hook code does not affect the communication flow of the program (because the communication code is generated and cannot be edited by the user), there is a greater possibility that this code can be used in another pattern. It may not always be possible to do this, though, because the hook code may use keywords or take advantage of pattern–specific features. However, this feature does provide better separation between the design pattern and the user code than earlier efforts.

We can allow the parallelism in a program to be hierarchically specified by allowing patterns to be substituted for the sequential components in a program. This composition can be compared to the Composite design pattern [16]. It provides a structured way of building complex program elements.

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

Currently, this research has not focused on how to provide a way of extending the set of existing patterns. There has been other work in this area, such as the DPnDP system [42]. We hope to use and continue the work started by this system.

We have already discussed openness extensively earlier in this paper. We will not discuss it further.

5.2.2 Programming

One of our priorities with the CO_2P_3S system is to reduce the probability of programmer error. Our approach is to remove the burden of writing the code for the parallel structure of the application from the user; in our case by automatic template generation. The user can concentrate on the application–specific code for the hook methods with the assurance that the underlying communication and synchronization structure is implemented correctly. This correctness guarantee also simplifies incremental tuning since the user starts the process knowing that the program structure is correct. The user can also regenerate this initial, correct solution at any time during tuning, if errors are introduced.

Providing the correct implementation of the design patterns in our PPS also allows us to reason about the semantics of the user program at the first layer of abstraction. For example, an object in a mesh computation should probably reference all of its neighbours. This reasoning can only be done if the programmer cannot violate the constraints of the selected design patterns by modifying the generated code.

In examining the language characteristics, we should emphasize that the CO_2P_3S architecture is intended to be independent of (object–oriented) language. As such, it should be possible to use the architecture for a variety of languages, satisfying the goal of using an existing, familiar programming language. The definition of this characteristic also includes the ideal of preserving the semantics and syntax of this programming language. We purposefully stray from this ideal, though, as we feel that preserving the semantics of a sequential language unnecessarily limits the potential concurrency. As a small example, consider programming languages that support run–time exceptions. To fully preserve the sequential semantics, it is necessary to execute every statement in order. However, concurrent execution could execute code that would not be executed in the event of an exception [45]. Without concurrent execution, computational parallelism is useless. A limited set of new keywords can provide a usable abstraction for parallel programming without greatly disturbing the rest of the language. We feel that the benefits of a well–planned abstraction that is properly integrated into the language can outweigh the risks of modifying a programming language. Nevertheless, we are only proposing these modifications at the intermediate code layer of our PPS with the understanding that at the native code layer, they will be translated to a standard programming language; in this case, Java or C++.

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

5.2.3 User Satisfaction

Without a fully implemented and mature system, most of the characteristics in this last category are impossible to evaluate. In particular, the support and usability objectives cannot be addressed. The performance objective is dealt with by our architecture for incremental tuning. Portability can be addressed by providing different implementations of the Native Code layer for different architectures.

5.3 Evaluating Other Parallel Programming Systems

We apply the criteria to the systems described in Section 2.1, even though not all are template–based. However, only the criteria that apply to a given system are evaluated. Also, we only mention the criteria that can be evaluated, based on public information about the systems, rather than speculating about them. We will concentrate on the first two criteria categories, leaving out the user satisfaction category. Most of the systems presented here have not performed detailed usability studies of their tools, making this objective difficult to evaluate. Further, it is difficult to evaluate and compare the performance of the different systems mainly due to the lack of a common benchmark suite and execution environment. We will mention the capabilities of the given systems with respect to portability and support tools where available.

5.3.1 FrameWorks

FrameWorks fails to meet most of the criteria. First, since a module explicitly calls other modules, any change in the templates must be mirrored in the code, violating the separation objective. Next, there were some non–intuitive constraints on the composition of the three templates needed to fully specify a module, resulting in poor usability and making the templates context–sensitive. The blocking `call` statement was a source of inefficiency, decreasing performance. FrameWorks fails the extendible criteria since there is no facility for adding new patterns to the system. It also fails the openness objective since users cannot access the run–time facilities. Finally, users have to rewrite portions of their sequential code to pack parameters into frames, which can be difficult if the original code relied on pointers. This limitation violates the non–intrusiveness objective. Combined with the `call` and `reply` statements, this limitation also violates the language objective.

FrameWorks does satisfy some of the criteria, though. The templates do exhibit some degree of independence, with the body template being independent of the input and output templates. Further, communication and synchronization were performed by the run–time system, providing some correctness guarantees.

5.3.2 Enterprise

Enterprise improved on FrameWorks in several ways, meeting more of the desired characteristics. First, an asset specifies the template completely, rather than the three part specification required by FrameWorks. This improved the usability of the system. Enterprise provided more templates for the user, increasing the utility of the system. The programming model uses (almost) sequential C code, satisfying the language objective. Further, the programming model removes the FrameWorks restriction of packing parameters into frames that cannot include pointers, improving upon the non–intrusiveness objective. Next, the decision for executing a procedure locally or remotely is made based solely on the meta–programming model. Since the meta–programming model is not reflected in the user code, the diagram and code can be modified in a more independent manner, which satisfies the separation objective. Enterprise assets are combined hierarchically through coercion, which meets the hierarchy criteria. Enterprise provides some correctness guarantees about the user program, ensuring that a program matches its meta–program at compile time and ensuring that the communication cannot result in a deadlock. Finally, Enterprise provides improved performance through its asynchronous procedure calls.

However, Enterprise still suffers from some limitations. It fails to meet the extendible criteria, providing no way of extending the set of templates available to the user. The programming model is still intrusive, since the programmer must remember to separate the invocation of a function from any accesses to the results to ensure good performance. Lastly, it fails the openness objective; users must write their programs using only the set of assets available, and cannot access the run–time system.

5.3.3 Tracs

The Tracs system fails to meet some of the criteria. The graph representing the application is not composed hierarchally. Tracs does not make any correctness guarantees unless the behaviours in the sequential code option are used. The largest limitation, though, is that Tracs fails to meet the non–intrusiveness and language objectives.

Tracs meets most of the remaining criteria. It achieves separation by directing messages to ports rather than the destination module. The code for a task only needs to be changed if new ports or services are added. The system is extendible, allowing a user to define and reuse architecture models.

5.3.4 Parsec

Parsec meets several of the criteria. First, it achieves separation by communicating through ports. It also provides some correctness guarantees about the overall structure of an application through the use of template code. Further, Parsec does not require changes to the chosen sequential language. Finally, it does contain some hierarchical composition for the definition of modules, but it is not clear if this composition also applies to the templates themselves. As a result, it is also unclear if the templates are also context insensitive.

Parsec fails with respect to several of the criteria. First, the system is not open, encapsulating the template in the virtual machine and keeping it from the user. The system does not provide a means of extending the set of available templates, which requires the implementation of new virtual machines. Finally, the explicit communication and message types in the process code is intrusive. The explicit message passing code also detracts from correctness since the user is responsible for properly implementing this code.

5.3.5 DPnDP

DPnDP meets all of the criteria except for non–intrusiveness and, to some extent, correctness. Separation is achieved by communicating through ports. Composition meets the hierarchy objective. The patterns are independent by their uniformity. DPnDP provides a method for introducing user–defined patterns into the system, meeting the extendibility criteria. The system is open. It further handles the pattern–specific communication, providing some correctness. Lastly, it uses an established language (C++) and does not modify the syntax or semantics of the language.

However, the explicit communication programming model violates non–intrusiveness. Further, the programmer marshals the data sent between modules, leaving the correctness of this important aspect of an application to the user, which is a violation of correctness.

6 Other Related Work

This section describes other work related to this research in both the design pattern and parallel programming fields.

6.1 Design Patterns

Design patterns can be described as repositories of design experience, introduced to the object–oriented community in the seminal work of Gamma *et al.* [16]. Patterns describe commonly occurring design problems and provide the outline of a solution that can be implemented for a number of problem contexts. This field is based on the observation that good designers do not start new designs from first principles but rather use experience with similar problems to guide them to a solution. Patterns are reuse at the design level.

To exploit design reuse, patterns define a vocabulary that designers can use to explain and document the structure of their programs. Documenting programs using patterns provides a concise description that is easily understood by those who understand the vocabulary. Patterns also facilitate analysis of programs by explaining the consequences of using the pattern, clarifying the benefits and drawbacks of the resulting design. In a community fluent in design patterns, it is possible to design at a level above individual objects and classes, facilitating the creation of larger and more complex designs.

To promote the exchange of patterns, a common description format has been created. The description specifies the following (taken from [15]):

Pattern Name A good, descriptive name for the pattern is essential if it is to be adopted into the common vocabulary.

Intent A description of the rationale behind the pattern and the design issues it addresses.

Motivation A scenario that illustrates an example problem that the pattern can be used to solve.

Applicability A description of the circumstances in which the pattern can be applied.

Participants A description of the objects and classes that are used in the pattern and their responsibilities.

Collaborations A description of the interaction between the objects and classes used in the pattern.

Diagram A graphical representation of the structure of the collaborating objects.

Consequences A discussion of the trade–offs involved in using the pattern.

Implementation A discussion of any issues that may be encountered during the implementation of the pattern.

Examples Real–life examples of the use of the pattern, usually from at least two separate problem domains.

See Also References to other related patterns, either patterns similar in structure and intent or patterns that can be used in conjunction with the current one.

However, patterns are not restricted to design reuse. They have been used as a teaching aid, demonstrating the principles of good object–oriented design to students.

Patterns can be classified along two axes: *purpose* and *scope*. Purpose describes what a pattern does. The scope of a pattern may be *creational*, *structural*, or *behavioural*. Creational patterns describe ways of creating objects. Structural patterns describe how objects and classes can be organized to achieve design goals. Behavioural patterns describe how objects and classes interact. The scope of a pattern indicates if the pattern is applied to objects or classes. Class–based patterns typically describe how inheritance relationships can be used, whereas object–based patterns describe the dynamic interactions between objects.

The work of Gamma *et al.* concentrated on patterns for sequential object–oriented programming. However, as concurrent and parallel programming enters the computing mainstream, new patterns are emerging to help programmers deal with synchronization, locking, and other programming difficulties in this area [38, 23].

We can also see pattern–based programming tools beginning to emerge. *Pattern–Lint* [36] is a tool that uses a combination of static and dynamic program information to verify that a program has correctly implemented a given design pattern. The intent is to ensure that a program follows its design since any deviations can make future maintenance much more difficult. The authors suggest the more proactive idea of generating code from design specifications to accomplish this task, which forms part of the basis for our pattern layer. This code may also include run–time assertions to further enforce compliance with the specified design. We do not include such code since the purpose of our open architecture is to allow the user to modify the generated code and possibly its structure.

A concrete example of code generation was implemented by Budinsky *et al.* [10]. The project provides a web interface that allows the user to generate correct code for the patterns in [16]. This code is downloaded by the user and modified for its exact purpose. In contrast, our approach does not allow the user to modify the generated code immediately, but instead provides hook methods that the user implements.

We should note that although the majority of work on design patterns has been done by the object–oriented community, patterns are not limited to this paradigm. The ideas are equally applicable to all programming paradigms. However, many of the current patterns take advantage of object–oriented features such as inheritance and polymorphism which makes them difficult to implement in non object–oriented languages.

6.2 Parallel Programming Libraries and Tools

Parallel programming systems are only one approach to writing parallel programs. This section discusses other approaches and the libraries and tools that implement them.

6.2.1 Parallel Programming Languages

The language criteria suggests that template–based parallel programming systems should use an existing commonly–used sequential programming language. However, new explicitly parallel programming languages are a useful outlet for research in parallel processing. New languages provide more flexibility for doing new research, allowing language developers to tailor the language to the abstractions they wish to provide. This can be accomplished by including constructs that mirror the desired programming model and eliminating constructs that may cause problems. Unfortunately, new languages have several serious drawbacks. They impose a steep learning curve on new users. More importantly, this approach prevents users from incorporating existing code into their parallel applications. Finally, as with sequential languages, the correctness of a program is the responsibility of the user.

Obviously, some researchers have found that the benefits of a new language outweigh the costs. Within this field, we can see a variety of approaches to creating a new language and a variety of features and abstractions that are included. A good summary of concurrent object–oriented and object–based languages and their features can be found in [31]. We can see a variety of approaches to all aspects of parallelism in these languages. Languages can be targeted to processes distributed over a network (Orca) or threads on a uni– or multi–processor (μC++). Concurrency may be created through an explicit `fork/join` model (Orca) or asynchronous method calls on annotated objects (Mentat). The parallel elements in a language may communicate through explicit message passing (Scoop), shared memory abstractions (Orca), or method invocations (Mentat). The design of the languages also demonstrates some of the available approaches. Mentat is an extension of an existing language, C++. Orca is a completely new language. Finally, Eiffel// tries to preserve the syntax of Eiffel but modifies the semantics of any class deriving from a special `PROCESS` class.

We have found one language, P^{3}L, that tries to use patterns to provide correctness. We discuss this language in some detail below.

P^{3}L P^{3}L (the Pisa Parallel Programming Language) [1] is an explicitly parallel programming language with dataflow semantics and structured, typed communication. The language defines a set of patterns the user can compose to create applications. The resulting program is executed by an *abstract machine*, similar to the virtual machines in Parsec. This abstract machine handles the low–level details of executing the program.

P^{3}L defines a set of patterns that are used to define the parallel structure of the application. The user can only specify the structure using these patterns; it is not possible to write programs that do not conform to the provided structures. The basic pattern is called a *sequential* construct, which is a sequential process containing code to execute. The code is an attached C++ code fragment specified by the user. The remaining patterns are common parallel design patterns, such as task farms, pipelines, and meshes. These patterns may be composed in a hierarchical manner by replacing any sequential construct with a pattern.

The overall programming model provided by the $\mathrm{P}^3\mathrm{L}$ language is based on dataflow semantics. Each process reads input from a typed input stream and places its output onto a typed output stream. Both streams appear as lists of parameters to the code fragments.

The most interesting aspect of this work is the compiler, which is responsible for generating the abstract machine used to execute a $\mathrm{P}^3\mathrm{L}$ program. The compiler consists of three stages: the *front end*, the *middle end*, and the *back end*. The front end compiles the pattern in the program into an intermediate call graph called a *construct tree*, an internal representation of the parallel structure. This stage also compiles the C++ code in the sequential constructs. The middle stage creates abstract process mapping structures based on the construct tree and an analytic cost model. The cost model and mapping are machine–dependent. This stage may modify the structure of the program if the cost model indicates the parallelism is unnecessary or wasteful. Finally, the back end generates the final configuration of the abstract machine, including the final mapping of processes to processors and instantiation of communication channels. The abstract machine itself is responsible for mapping processes to processors, communication, and scheduling, removing these details from the user.

The compiler also presents an opportunity for parallel programs developed in $\mathrm{P}^3\mathrm{L}$ to be portable across a number of hardware architectures by generating different abstract machines. Further, the cost models can be used to optimize the program for the given hardware, maintaining performance.

$\mathrm{P}^3\mathrm{L}$ meets most of our criteria for evaluating template–based systems. It achieve separation of application code from the template since the code in the sequential constructs uses the input and output streams for communication, rather than explicitly referencing the other constructs. A program can be composed hierarchically. The supplied templates are independent of one another, and may be used in any combination. The abstract machine does make some correctness guarantees, automatically performing scheduling, mapping, and communication. Lastly, the system does partially meet the language criteria by allowing the user to write their application in C++.

However, $\mathrm{P}^3\mathrm{L}$ does not meet the extendibility criteria. The user cannot add new patterns into the language. The language does not meet the openness objective, as the user cannot access the abstract machine. It also partially fails to meet the language criteria since the structure of the program is specified in a new, unfamiliar programming language and is further written as code fragments and not as a normal program. Finally, the new language and dataflow model are intrusive.

6.2.2 Parallel Programming Libraries

A common approach to parallel programming is based on communication libraries such as MPI (Message Passing Interface) [12], PVM (Parallel Virtual Machine) [43], and NMP (Network Multi–Processor) [25]. These libraries provide basic message passing primitives that allow the user to explicitly send bytes between different processes. The principle power of this approach is flexibility; the user is free to implement any communication structure that is appropriate for the application. If the communication system supports a heterogeneous environment, the user may also take advantage of any special capabilities of the different machines.

An alternative library–based approach uses a library to provide high–level parallel programming constructs for the programmer. Some libraries, such as ABC++ [30], are written as higher–level constructs on top of a message–passing library. ABC++ is based on active objects in a distributed system. Communication is based on synchronous or asynchronous method invocations, the latter synchronized by explicitly created future objects. ABC++ also provides a distributed shared memory abstraction called parametric shared regions. The user is free to implement any parallel structure using this library.

In contrast, PUL (Parallel Utilities Library) [8] is a set of libraries implementing different parallel programming structures or supporting utilities. PUL is implemented on top of its own portable message passing layer CHIMP. PUL includes libraries for implementing task farms (PUL–TF), decomposing meshes (PUL–RD) and then operating on them (PUL–SM), accessing files (PUL–GF and PUL–PF), and group communication (PUL–EM). PUL libraries, where applicable, provide two different interfaces: *skeletal* and *procedural*. The skeletal interface is a template–based approach, with the user filling in missing pieces of a skeleton that encapsulates both communication and the structural code of the application. The procedural interface provides a flexible set of library primitives the user may use to express programs, hiding only the communication between the different application elements.

In return for the flexibility of the library–based approach, the user becomes solely responsible for the low–level details and correctness of the program. Messages must be properly marshaled and unmarshaled, synchronization is explicit, and the user may be responsible for placing processes on the appropriate processors. There are also serious design drawbacks. The parallel structure of the application is coded into the program, making it difficult to modify. Finally, while the program may compile and execute on different hardware, getting the best performance out of the existing structure will require some porting effort.

Our pattern–based efforts provide correctness and usability in exchange for a loss of flexibility and performance. We feel this tradeoff will be acceptable for a broad user base who will appreciate improved performance so long as the programming costs do not rise significantly. However, the openness of our system may also address some of the concerns of high performance users, who can use the higher layers of our system to prototype their applications and use the generated code as a starting point to creating their own programs.

6.2.3 Parallel Programming Frameworks

Another approach to parallel programming is *frameworks*, a concept similar to templates and that can be related to design patterns. A framework is an extendible set of classes that implements the basic structure for solving problems, usually in a specific domain [21]. This solution may or may not implement some templates or patterns, although doing so provides a good way of describing how a user can interact and extend the framework. A user uses a framework by specifying a set of components or objects that the computational portion of the framework uses. If the desired components do not exist, a user extends the framework by subclassing existing classes and overriding their behaviour, still taking advantage of structural code.

Frameworks differ from patterns by providing a solution for a narrow problem domain and by providing code for that solution. Patterns, in contrast, describe generic solutions to problems that may be used across different problem domains. For instance, the Model–View–Controller (MVC) framework for graphical user interfaces uses the Observer pattern to notify its components of any changes so that the display is always up to date. However, the Observer pattern is more general and can be used for other problems.

Frameworks differ from templates by providing a variety of existing components that can be used to construct an application. For simple applications, a user may only have to create an instance of the framework with these components rather than inserting any new code. Frameworks are also unlike templates in that the problem domain is usually restricted whereas our templates provide a general structure for user programs. Frameworks are similar to templates by encapsulating the structure of the problem. This allows a framework to use an optimized implementation of its structure based on its execution environment.

There are several examples of frameworks in the literature, including Khoros [22] and POOMA (Parallel Object–Oriented Methods and Applications) [34]. Both systems are frameworks that operate within a fixed problem domain. Khoros defines operations for image processing and data visualization that are linked together using a visual programming language that is similar to the graphical tools discussed in Section 2.2. POOMA provides a layered framework for data parallel computations for scientific problems and simulations, providing classes for fields, vectors, matrices, and particles. A user may work at any given layer to extend the functionality of the framework.

Although useful for more restricted problem domains, frameworks cannot easily be applied to the more general problem of writing parallel programs. User programs do not follow a standard interface, making it difficult to re–use the computational engine of a generic framework. We must either introduce a potentially intrusive interface that the user must follow or incur the costs of some form of reflection to allow general interaction between classes[6]. Frameworks may be open, allowing a user to subclass the structural code to modify it as appropriate. However, the user will also need to write more code to inform the framework of the new classes and make sure they are used. In contrast, our generated code allows a user to change the structure directly rather than having to learn the structure of the framework.

6.2.4 Parallelizing Compilers

The ideal solution to the problem of parallelizing programs would be a compiler that takes a sequential program as input and generates an optimal parallel program while preserving the sequential semantics of the original code. There have been some attempts to develop just such a compiler, such as SUIF [18], Parafrase–2 [32], Paradigm [2], `javar` [6], and `javab` [7].

[6]Reflection in object–oriented languages is a mechanism that allows the internals of an object to be examined, and usually includes a facility for invoking a method based on a name and signature.

The principle weakness with parallelizing compilers is that a parallel version of an optimal sequential algorithm does not always represent an optimal parallel algorithm. The quicksort algorithm suffers from this weakness. While quicksort is the best overall sequential sorting algorithm, the parallel version of the algorithm is limited to a speedup of 5 to 6[7][37]. In contrast, parallel sorting by regular sampling [37] is much more scalable, uses available processors more effectively, and achieves better results.

Another weakness of many parallelizing compilers is that they cannot detect coarse–grained parallelism. Many concentrate on loop parallelism, using data dependence analysis to find independent loop iterations that can be executed in parallel.

Finally, some features in modern computer languages inhibit general compiler optimizations, including those used for parallelism. For instance, language features such as exceptions and dynamic method binding in languages like Java and C++ cause problems for optimizing compilers [9]. The problems with exceptions appear in the `javar` and `javab` projects, neither of which preserves sequential semantics with respect to exceptions. `javar` performs source code translation on Java code to parallelize loops regardless of any exceptions that may be thrown. However, if the code within a parallel loop throws an exception, the user cannot make any assumptions about which loop iterations have completed because of the non–deterministic execution of the loop. `javab` performs translations on Java bytecode and only parallelizes loops that can be proven exception–free. Exception–free code is code that either contains no instructions that can throw exceptions or code where the conditions are such that the instructions that may thrown exceptions will execute correctly (such as array accesses where the index can be statically shown to be within bounds). However, this guarantee does not extend to virtual machine failures (also modeled as exceptions) or any linking exceptions encountered while executing the code.

6.2.5 User–directed Parallelizing Compilers

A slightly different approach to the parallelizing compilers described in the previous section is an approach we will call *user–directed* parallelizing compilers. The user inserts directives into the application code to instruct the compiler on how to parallelize the program. This approach is taken in High Performance FORTRAN [14] and Parallel Application Management System (PAMS) from Myrias [27]. In both cases, the user inserts directives that appear as comments in the code to indicate data distribution, parallel loops or `pardos`, and parallel blocks or `cobegin/coend` pairs. An alternative is interactive compilation, as demonstrated by the `javab` tool. The tool will ask the user for help in recognizing exception–free code blocks if static analysis is inconclusive.

By not specifying the parallelism in the application code, these systems take advantage of a user's knowledge of the sequential language. Also, the specification of the parallelism is separated from the program code and can be modified easily. Lastly, if the sequential semantics of the language are maintained, the

[7]Speedup is defined as $\frac{T_s(n)}{T_p(n)}$, where $T_s(n)$ is the time taken to execute a program sequentially and $T_p(n)$ is the time taken to execute the same program in parallel.

program can be tested sequentially.

The principle weakness with this approach is that the user is typically responsible for the correctness of the resulting application. The compiler may not verify that there are no dependencies in the specified parallelism. Further, the systems are not open; the user can only use the directives to specify parallelism. Lastly, this approach is still parallelizing sequential code and suffers from the problems identified in Section 6.2.4.

7 Proposed Research

7.1 An Open Parallel Programming System

The description of the CO_2P_3S system leads to a clear avenue for research in the construction and verification of the presented architecture.

The programming benefits of an architecture that provides multiple user–accessible layers of abstraction is still an open question [17] that this work may help answer. This research will also advance the present work in pattern–based parallel programming systems by maintaining a high–level programming model while preserving the openness and extendibility found in some existing systems. These systems sacrificed the programming model to achieve their other benefits; we feel this sacrifice is unnecessary.

We also need to find a way of presenting these abstractions to the user without sacrificing the usability of the resulting system. It is crucial that the user be able to work with code at any layer in our system.

It is also critical that the system is capable of providing acceptable performance at all levels. This research cannot be considered a success if the resulting PPS cannot be used because it cannot provide users with any performance benefits. After all, improved performance is the primary reason for parallel processing.

This research can be done by creating a prototype of the CO_2P_3S with a small number of templates. Further, the compiler transformations may be implemented as a preprocessor rather than creating a language that would require compiler support. If desired, a compiler for a new language can be implemented in the future. This language can use another high–level language as an intermediate language. The intermediate language would serve as our native code layer.

Another possible result of this research is a hierarchy of generalized patterns for parallel programming. We may find that some patterns can be related to others, either semantically or structurally. This kind of hierarchy may be able to guide our implementation of CO_2P_3S as well as provide a clear organization of parallel patterns for potential users of the system. Further, it may be possible to organize other patterns in a similar fashion, making it easier to relate patterns to one another and use them.

We should emphasize that our solution is not necessarily a complete solution to parallel programming in general. Rather, it will demonstrate that it is possible to provide an open environment that can integrate several different abstractions for writing parallel programs.

7.2 Pattern Composition

A more general question relates to what properties of a pattern–based program are preserved by composition. First, we need to define exactly what is meant by composing two patterns and specifically how they interact. However, we can already infer some basic results from earlier work in pattern–based systems. For instance, we know that the composition of deadlock–free patterns is not deadlock–free, a result provided by FrameWorks. However, results for the correctness of an algorithm or the correctness of synchronization are unclear.

We would start this work by deriving results for a particular set of patterns. It may then be possible to generalize the results.

7.3 Explicitly Parallel Object–Oriented Languages

On the surface, adding concurrency to object–oriented languages appears to pose no serious difficulties. The paradigm refers to sending messages to objects, rather than invoking methods, so it should be straightforward to translate these lightweight messages into heavyweight communication. Unfortunately, synchronization of objects poses serious problems, the foremost being the *inheritance anomaly* [26]. The inheritance anomaly notes that if synchronization code appears as part of the methods in a class, it may be necessary to re–implement the functionality of some or all of these methods in any subclass that modifies the synchronization constraints. Even if this redefinition is unnecessary, a programmer still needs to understand enough about the implementation of a superclass to know that such changes are not required.

In the intermediate code layer of the proposed system, we will encounter this problem. The solution to this problem remains an open question.

8 Conclusion

A survey of previous research into parallel programming was reviewed. This survey included tools directly analogous to the proposed CO_2P_3S system as well as other common techniques for writing parallel programs. Within the domain of our new project, three avenues of research were presented: creating an open pattern–based programming system with a high–level programming model, composing patterns, and explicitly parallel object–oriented programming language development. Each avenue provides open questions, and the proposed work represents new work in parallel programming and design pattern research.

Parallel programmers have been following the basic principles of design patterns for a long time. Although the object–oriented community has been the most visible group to embrace design patterns, parallel programmers were likely the first group to make extensive use of them. However, parallel programmers have tended to avoid object–oriented technology because the benefits come at the expense of performance, a trade–off that has only recently come to be seen as appropriate. With the introduction of object–oriented languages and techniques into parallel programming, it is only natural to attempt to take advantage of the recent work on design patterns and apply it to high performance computing. We hope the work in this thesis can further this goal.

References

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

[2] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed–memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[3] A. Bartoli, P. Cosini, G. Dini, and C. A. Prete. Graphical design of distributed applications though reusable components. *IEEE Parallel and Distributed Technology*, 3(1):37–51, 1995.

[4] A. Beguelin, J. Dongarra, A. Giest, R. Manchek, and K. Moore. HeNCE: A heterogeneous network computing environment. Technical Report UT-CS-93-205, University of Tennessee, August 1993.

[5] A. Behuelin and G. Nutt. Visual parallel programming and determinacy: A language specification, an analysis technique, and a programming tool. *Journal of Parallel and Distributed Computing*, 22(2):235–250, 1994.

[6] A. J. C. Bik and D. B. Gannon. Exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, 1997.

[7] A. J. C. Bik and D. B. Gannon. `javab` – a prototype bytecode parallelization tool. Technical Report TR 489, Department of Computer Science, Indiana University, July 1997.

[8] R. A. A. Bruce, S. Chapple, N. B. MacDonald, A. S. Trew, and S. Trewin. CHIMP and PUL: Support for portable parallel computing. In *Proceedings of the Fourth Annual Conference of the Meiko User Society*, April 1993.

[9] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, 1997.

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

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

[12] J. Dongarra, S. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[13] D. Feldcamp and A. Wagner. Parsec – a software development environment for performance oriented parallel programming. In *Proceedings of the Sixth Conference of the North American Transputer Users Group (NATUG 6)*, pages 247–262, May 1993.

[14] High Performance FORTRAN Forum. High performance FORTRAN language specification version 2.0. Technical Report CRPC–TR92225, Center for Research on Parallel Computation, Rice University, January 1997.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object–oriented design. In *Proceedings of the 7th European Conference on Object–Oriented Programming (ECOOP'93)*, pages 406–431, July 1993.

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

[17] M. Haines, G. Benson, and K. Langendoen. On building efficient substrate software. In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97)*, pages 325–333, June 1997.

[18] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[19] R. Halstead. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[20] P. Iglinski, S. MacDonald, C. Morrow, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron, and D. Woloschuk. Enterprise user's manual version 2.4. Technical Report TR 95–02, University of Alberta, January 1995.

[21] R. E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[22] Khoral Research, Inc. *Introduction To Khoros 2.0*, 1994. http://www.tnt.uni-hannover.de/soft/imgproc/khoros/khoros2.

[23] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley, Reading, Massachusetts, 1997.

[24] S. MacDonald, J. Schaeffer, and D. Szafron. Pattern–based object–oriented parallel programming. In *Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, December 1997.

[25] T. A. Marsland, T. Breitkreutz, and S. Sutphen. A network multi–processor for experiments in parallelism. *Concurrency: Practice and Experience*, 3(1):203–219, 1991.

[26] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, Cambridge, Massachusetts, 1993.

[27] Myrias Computer Technologies Corp. *PAMS On–line Documentation*, 1995. http://www.myrias.com.

[28] P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming language. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 167–177, July 1992.

[29] P. Newton and J. Dongarra. Overview of VPE: A visual environment for message–passing. In *Proceedings of the 4th Heterogeneous Computing Workshop*, April 1995.

[30] W. G. O'Farrell, F. Ch. Eigler, S. D. Pullara, and G. V. Wilson. ABC++. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, chapter 1, pages 1–42. MIT Press, Cambridge, Massachusetts, 1996.

[31] M. Philippsen. Imperative concurrent object–oriented languages. Technical Report TR–95–050, International Computer Science Institute, August 1995.

[32] C. D. Polychronopoulos, M. B. Girkar, M. Resa Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase–2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 39–48, August 1989.

[33] M. Rao, Z. Segall, and D. Vrsalovic. Implementation machine paradigm for parallel programming. In *Proceedings of Supercomputing '90*, pages 594–603, November 1990.

[34] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M Srikant, and M. Tholburn. POOMA. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, chapter 14, pages 547–587. MIT Press, Cambridge, Massachusetts, 1996.

[35] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

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

[37] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[38] A. R. Silva. Framework, design patterns and pattern language for object concurrency. In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97)*, pages 1024–1033, June 1997.

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

[40] A. Singh, J. Schaeffer, and D. Szafron. Experience with template–based parallel programming. *Concurrency: Practice and Experience*, 1997. To appear.

[41] S. Siu. Openness and extensibility in design–pattern–based parallel programming systems. Master's thesis, Department of Computer and Electrical Engineering, University of Waterloo, 1996.

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

[43] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

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

[45] A. Zubiri. An assessment of Java/RMI for object–oriented parallelism. Master's thesis, Department of Computing Science, University of Alberta, 1997.