University of Alberta

Library Release Form

Name of Author: Steven MacDonald

Title of Thesis: From Patterns to Frameworks to Parallel Programs

Degree: Doctor of Philosophy

Year this Degree Granted: 2002

Steven MacDonald
1410–8210 111th Street
Edmonton, Alberta, CANADA
T6G 2C7

Date: ______________

University of Alberta

From Patterns to Frameworks to Parallel Programs

by

Steven MacDonald

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **From Patterns to Frameworks to Parallel Programs** submitted by Steven MacDonald in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Jonathan Schaeffer
Supervisor

Duane Szafron
Supervisor

H. James Hoover

Eleni Stroulia

Michael Carbonaro

Douglas Lea
External Examiner

Date: ____________________

Abstract

Parallel programming offers potentially large performance benefits for computationally intensive problems. Unfortunately, it is difficult to obtain these benefits because parallel programs are more complex than their sequential counterparts. One way to reduce this complexity is to use a parallel programming system to write parallel programs. This dissertation shows a new approach to writing object–oriented parallel programs based on design patterns, frameworks, and multiple layers of abstraction. This approach is intended as the basis for a new generation of parallel programming systems.

A critical evaluation of existing parallel programming systems is presented. This evaluation is based on a set of 13 characteristics of an ideal pattern–based parallel programming system. Based on the results of this evaluation, the PDP process was created. This process provides multiple layers of abstraction to support the complete application development cycle. The first layer supports pattern–based parallel programming by generating object–oriented frameworks from a design pattern description of the application structure. The generated code is hidden from the user to ensure correctness. Lower layers gradually provide access to the generated structural code and other low–level

facilities.

The CO_2P_3S parallel programming system is an example of a tool that supports the PDP process. It generates multithreaded Java frameworks for a set of supported design pattern templates. The utility of this tool is demonstrated by four example programs which obtain reasonable speedups after being written using CO_2P_3S. Further, the results of a usability experiment are presented. The experiment compared parallel programming using CO_2P_3S against programming in non–CO_2P_3S Java with threads. The results show that CO_2P_3S users write less application code than their Java counterparts, and that this code is less complex.

Acknowledgements

I have heard that if you don't acknowledge your supervisors first, they won't write nice reference letters for you. So, I'd like to thank my wife Trudy for her love and patience. I also have to thank my family far all of their support.

Okay, now a big thank you to Jonathan and Duane for all of the guidance and support over the past eight (!) years. They have been great collaborators and played an invaluable role in making me the researcher I am today. They have also been great friends.

Thank you to the other members of my examining committee: Jim Hoover, Eleni Stroulia, Mike Carbonaro, and Doug Lea. I appreciate all of the effort you expended on my behalf.

I absolutely must acknowledge the work of Steven Bromling. He was responsible for much of the design and all of the implementation of the CO_2P_3S user interface. Through his efforts, CO_2P_3S advanced from an idea on paper to a tool I can see and use. I also have to thank the other members of the CO_2P_3S research group: Kai Tan and John Anvik. Together with Steve Bromling, they've all tackled interesting problems and improved this work.

I have to thank the students in CMPUT 425 who served as the subjects in my usability study. I can only assure them that, contrary to what they may have thought at the time, I was not trying to make their lives miserable.

I also have to thank the people who helped me run my experiments. The Research Computing Support Group at CNS, notably Ron Senda and Wally Lysz, gave me access to the Origin systems. John Bartoszewski and Rod Johnson let me run programs on their Sun server.

There have been a large number of people I've run into during my stay here. All of them have made their mark. Some of the notables, who have always been good for some conversation or a bit of fun, are: Diego Novillo, Nelson Amaral, Paul Lu, Ernie Novillo, Chris Dutchyn, Cam Macdonell, Maria Cutumisu, and Darse Billings. Elmer, Sean, Andrew and Eric also deserve mention. And, of course, everyone back in Ontario who wondered if I was ever coming back.

Of course, I big thanks to the D&D group for all of the amusement on Wednesday nights: Gord, John, Jonathan, Mark, Dave B., Wade, Ian, Duane, and Dave W.

And finally, thanks to everyone else. I'm not forgetting you. I've just run out of room, and there are simply too many of you.

A final note: you are all encouraged to read [3] for your own benefit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction and Motivation

Parallel programming offers the possibility of large performance benefits for computationally–intensive problems for applications in areas such as computational biology, physics, chemistry, and computer graphics. Computational problems from these domains can take hours, days, or even weeks of processing time on a single processor. The turnaround time for these large programs cannot be improved significantly by simply buying a faster computer or more memory. Instead, this improvement must be achieved by splitting the problem over multiple processors, each solving a different part of the overall problem in parallel.

Unfortunately, it is often difficult to obtain the performance benefits of using multiple processors. Expert programmers must design new, highly concurrent algorithms. These new algorithms must address concerns such as communication and synchronization, which are not present in sequential programs. These new algorithms must then be implemented properly, which is difficult because they are more complex than sequential programs. Concurrent algorithms can also be more difficult to debug because of problems such as non–determinism. Finally, these algorithms must be tuned for efficiency, which may require detailed knowledge of the hardware and software architecture of the target platform. Overall, the process of writing a parallel program is usually complex and error-prone.

Much like the sequential domain, though, many of the strategies for solving computationally intensive problems share a common structure. This common structure can be abstracted out to capture experience in building parallel applications that can be shared among programmers. For example, parallel programmers have a set of common structures that they start with when creating a new program, such as fork/join parallelism, pipelines, work farms, and meshes. These basic design structures are not application–specific, but rather can be used to solve a broad range of problems. In sequential object–oriented programming, this idea is called a *design pattern* [37].

Design patterns name, describe, and evaluate solutions to recurring problems encountered in the design of object–oriented programs. A pattern's name provides a common vocabulary so programmers can quickly exchange relevant design information. The description of a pattern provides a sketch of the solution structure in a generic form. This generic form can be adapted for use in any number of problem domains. The evaluation covers several topics, including when the pattern is applicable, which aspects of the design are improved by applying the pattern, which aspects of the design are degraded by applying the pattern, and alternative structures for the pattern that may be useful for some problems.

While design patterns capture design experience, they fail to address code reuse; a pattern must be reimplemented each time it is used. A programmer may be able to find an older program that uses the pattern and can try to reuse that code, but the pattern structure may contain application–specific code that must be identified and removed. Further, the application–independent structural code may need to be modified for the new program. This process is time consuming and error–prone.

However, research in sequential object–oriented programming also provides a solution to this problem. In this case, we can use *object–oriented frameworks* to reuse the application–independent structural code [55, 37, 54]. One possible implementation of a framework is a set of abstract classes that provide the application–independent class and object structure of an application in a specific domain, such as a graphical user interface. This structural code defines the flow of control through the objects, often taking over control of the execution of the program. This flow of control is written in terms of *primitive* or *hook methods* that are called from the structure but not defined by it. A programmer creates an application by subclassing the abstract framework classes and overriding the hook methods with an application–specific implementation. This is in direct contrast to a library system, which provides an application–independent implementation of a set of primitive methods and requires the programmer to create the application structure using these methods. A programmer using a framework reuses not only the application–independent structural code, but also the complete design of the application.

This is not the only form of an object–oriented framework. Other frameworks also provide a library of pre–built components written by the framework developer. These components are subclasses of the abstract framework classes that provide implementations of hook methods that are common to different applications. The programmer can create and use these classes rather than writing their own subclasses. Another alternative is *pluggable objects*, in which a pre–built component is parameterized with application–specific information [55]. The pluggable object implements the hook methods required by the framework using the application–specific objects as collaborators. For example, a GUI framework may provide a pluggable view component that accepts a menu in its constructor, which is displayed when the correct hook method is invoked. Pluggable objects are easier for programmers to use than creating framework subclasses provided the parameter set is not overly complex. Over time, most frameworks evolve from requiring subclasses to providing components and pluggable objects [88]. As well, some frameworks do not monopolize the flow of control for the complete application. Instead, the framework provides the structure of a computation that the programmer specializes with the hook methods. This computation can be launched by the programmer and returns a result when it is finished.

Regardless of its form, the main benefit to a framework is that it can reduce the effort required to build a program. This savings is mainly the result of writing less code. Rather than develop a complete application, the existing framework structure is specialized for the particular problem. To obtain this benefit, the programmer must learn the framework design in order to understand how write a program with it. However, this one–time investment to learn the framework can be amortized over the many programs that can be built using it.

Frameworks can be applied to the parallel programming domain. The application–independent code in the abstract classes implements the parallel structure. Writing an application would require the programmer to implement a small set of hook methods. In the parallel domain, though, there is the possibility of using the frameworks to reduce the possibility of user error. The most difficult parts of implementing a parallel program are the communication and synchronization code. However, this code is part of the structure of a parallel program and can be placed into the abstract classes of the framework that are not modified during application development. Since all of the the parallel aspects of the application are now addressed by the framework code, the application–specific hook methods can be sequential code, possibly taken from an existing sequential code base.

This dissertation concentrates on applications that use parallel programming to minimize turnaround time. As such, performance issues cannot be ignored. Specifically, the code generated for a framework is intended to be generic so that it can be applied to many different applications. As a result, there may be overhead in the framework that is not needed for a specific application. This overhead may limit the performance that can be obtained. There must be some way of tuning the structure for better performance. This should be done in a controlled manner, gradually exposing the minimum level of detail needed to tune the performance. At the highest level, the details are limited to the implementation of the framework. At the lowest level, the complete source code, including all libraries used to support the provided abstractions, should be available. The programmer can select an appropriate level so that only those details relevant to the tuning problem are exposed.

Once again, there is a known solution to this problem: multiple programming layers. This idea has been used to separate concerns and hide unnecessary detail in many problems, such as computer network protocols. Applying layers to the tuning process allows a programmer to focus on those details that will improve the performance of an application without getting overwhelmed by other, unnecessary details.

1.2 Contributions

This dissertation explores a design–pattern–based approach to parallel programming. The approach also incorporates frameworks for a set of supported patterns and programming layers to provide multiple abstractions for program development and performance tuning. This approach is the result of a critical examination of existing parallel programming systems research based on the 13 criteria for an ideal template–based (pattern–based) parallel programming system. The contribution of this dissertation is to demonstrate that it is possible to construct a parallel programming system for building general parallel applications that:

1. Provides a layered programming model with multiple user–accessible abstractions for writing parallel programs. The breadth of existing parallel programming models shows that no single abstraction applies to the complete task of writing a parallel application. Instead, this task can be split into its constituent parts, each with a relevant abstraction that eases the task at hand by hiding unnecessary details.

2. Supports a design–pattern–based approach to parallel programming. Rather than building a complete application by hand, the programmer can start with a set of existing parallel program structures. This work shows a method of using design patterns in a software development system. In particular, it is important to note that a pattern is not a single design solution, but rather a family of related solutions, each sharing the same basic structure but with some variation to tailor the pattern to a given problem. This shows that it is possible to express this family in a programming system based on patterns. Also, it is possible for the patterns to be composed, as an application can consist of more than one pattern.

3. Generates a complete, correct structural framework for a parallel design pattern. All current parallel programming systems provide some form of abstraction for writing programs, but still leave it to the user to write part of the parallel structure (such as the exchange of messages between processors). However, the parallel structural code is the most difficult and error–prone part of a program to write and debug. Thus, this dissertation describes tool support for the generation of the parallel structure of a application. In generating a correct implementation of the structure, the user is saved the effort of writing and debugging this code, which eliminates an important class of potential errors in a parallel program.

4. Provides access to the parallel structural code so that it can be changed, to modify its structure and to improve its performance. Current parallel programming systems provide a single programming model that must be used for all programs. If the model cannot efficiently implement the desired program, there is no provision for bypassing the model. This limits both the range of problems that a system can efficiently solve and the performance of any program written with that system. This dissertation explores an avenue for gradually exposing the structure of a program and run–time support provided by the system to address these two problems.

5. Allows sequential code to be reused in a parallel application. Because all parallel aspects for a given pattern are part of the structural framework code, the application–specific parts of a program are written as sequential code. This sequential code can be taken from an existing code base.

6. Reduces the possibility of programmer errors during parallel program development. Traditionally, this means that the programs written using a parallel programming system should exhibit some correctness guarantees, such as freedom from deadlock. This work shows that tool support can also reduce the probability of errors when developing applications using object–oriented frameworks.

The result of this research is the CO_2P_3S[1] parallel programming system. This system is based on three layers of abstraction. At the first layer, a programmer selects a parallel structure from a set of *design pattern templates*, which are then customized using a set of parameters. The customized pattern template is then used to generate multi–threaded Java framework code, to which the user adds application–specific hook method implementations. The parallel structural code of the framework is encapsulated so that the user cannot introduce errors. The next layer exposes the framework structure using high–level constructs, so that this structure may be modified and optimized. The final layer provides access to the implementation of all of the abstractions in the framework so that they may be tuned for the execution environment.

An implicit goal in all parallel programming systems is usability; it should be possible for real users to build working parallel programs that achieve performance gains. However, with few exceptions [99, 96, 112], usability is not measured by researchers in the parallel programming community. Usability claims are typically based on anecdotal evidence provided by the developers of the system. A further contribution of this dissertation is the results from a usability study on CO_2P_3S, which compare parallel program development in CO_2P_3S against parallel program development in Java.

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

The end result of this work is the generalization of the CO_2P_3S application development model into the Parallel Design Pattern (PDP) process. This process is a tool–independent methodology that can be used as the basis for pattern–based parallel programming systems, of which CO_2P_3S is just one example. This process presents a series of abstractions aimed at improving the state of parallel programming systems research by providing both high–level programming models and low–level tuning capabilities to the complete process of creating an application. Further, the high–level model provides tool support for the most difficult aspect of writing a parallel program, creating a correct implementation of the program structure. Programming systems that are built using the PDP process will provide a flexible environment for developing parallel, object–oriented programs.

1.3 Organization

Chapter 2 discusses the 13 ideal characteristics of a pattern–based programming system and uses them to evaluate a representative sample of existing parallel programming libraries, systems, and languages. It also discusses other work in this area, including design patterns, pattern languages, frameworks, and object–oriented modeling languages.

Chapter 3 presents the PDP process in detail. The chapter uses an example program implemented using CO_2P_3S to concretely demonstrate the process from the perspective of a user building a program. The chapter also discusses issues that must be addressed by tool developers creating a system based on the PDP process.

The CO_2P_3S parallel programming system is presented in more detail in Chapter 4. Chapter 5 shows results from several example programs implemented with CO_2P_3S. Chapter 6 examines the results of the usability study performed on CO_2P_3S.

Chapter 7 gives the conclusions and suggests avenues for future research.

Appendix A provides the complete user code for the reaction–diffusion example used in Chapter 3. Appendices B and C provide more detailed information on some of the patterns supported by CO_2P_3S. Specifically, Appendix B describes the format for our pattern description. It is based on the format from [37]. Appendix C describes the Two–Dimensional Mesh pattern. Appendix D contains all reference material provided to the participants of our usability study. Appendices E and F are background material describing sequential and parallel mesh computations respectively, which were distributed to the subjects in our study. Finally, Appendix G provides more information on choice points, which are used as a measure of application code complexity in Chapter 6.

1.4 Publications

The format for the review of existing parallel programming systems in Chapter 2 first appeared in the Journal of Parallel and Distributed Computing [69]. This paper also contains the first description of the PDP process.

CO_2P_3S has been described in several additional conference publications [66, 67, 68]. The system has changed considerably since the first publication, and continues to evolve as new research is conducted.

Chapter 2

Related Parallel Systems Research

There has been considerable interest in developing parallel programming systems and tools for some time. This interest stems from the need to reduce the complexity associated with parallel programming, and the desire of programmers and users to utilize the multiprocessor systems that are now more readily available.

Section 2.1 evaluates related parallel systems research, including other parallel programming systems, libraries, and tools. Given the large amount of research in this field, this survey does not cover all systems, but rather examines a selection of work that is representative of the range of research that has been done. The basis for evaluation is the set of ideal characteristics of a pattern–based parallel programming system identified in [96]. These characteristics will show the strengths and weaknesses of earlier systems. Chapter 4 evaluates CO_2P_3S using these ideal characteristics to show how we have maintained the strengths of earlier research systems while addressing some of their weaknesses.

Section 2.2 discusses other relevant research, mainly in design patterns and pattern languages, that is also relevant to this work. Section 2.3 briefly surveys other issues related to frameworks, such as documentation, composition, and instantiation. Finally, Section 2.4 presents a small selection of modeling languages for building object–oriented programs in other program domains. Some of these languages use a layered approach to separate different programming and modeling concerns, and others have been the target of tool support that generates code from the user–supplied model.

2.1 Evaluating Parallel Programming Systems Research

To evaluate current parallel programming systems research, this dissertation uses the ideal characteristics of pattern–based parallel programming systems from [96]. These characteristics arose from insights and experience in building and using template–based parallel programming systems, which were the pre–cursors to the CO_2P_3S pattern–based approach.

The characteristics are broken up into three categories. Each category emphasizes a different aspect of a pattern–based system. Each characteristic is given a short name, provided in parentheses, that is used to refer to it throughout the rest of this dissertation. Each characteristic is described and followed by a discussion of existing research with respect to that characteristic.

2.1.1 Structuring the Parallelism

This category deals with the specification of parallelism for a user application. There should be as few restrictions as possible.

1. Separation of Specification (*Separation*): There should be a clean separation between the parallel structure of a program and the application code. This key feature allows both parts of a parallel program to evolve independently. It also allows for rapid prototyping of programs, and permits users to quickly experiment with alternative parallel structures.

   Code libraries fail to meet this concern. In all code libraries, the structure of the parallelism is embedded directly in the application code via library calls. This structure can only be changed by modifying the code, which may involve significant programming effort. For instance, for message–passing libraries such as PVM [38] and MPI [98], the parallel structure is dictated by the communication structure. A program using pthreads [103] or Java threads [7] must explicitly create the threads that will form the parallel structure of the program. Systems such as PUL [20] (in procedural mode) and archetypes [71] provide libraries with primitives for implementing specific program structures such as mesh computations and task farms. For instance, the libraries provided for mesh computations may include global reduction and boundary exchange communication primitives. In this case, the parallel structure is dictated by both the insertion of the library calls and the selection of the library.

   Most parallel programming languages and language extensions also suffer from the same problem by using extra syntax for specifying parallelism in the program code. For instance, Mentat [42], ABC++ [80, 6], JavaParty [83] and Orca [9] support task parallelism by allowing a programmer to annotate certain classes to execute in parallel. Mentat also includes an extra statement to return a future [44] for the result of a parallel method to the caller before that method has finished executing. Braid (an extension of Mentat) [109] and Orca [48], as well as the Java–based languages Spar [106] and Titanium [116], support data parallelism by including parallel loops (with a `forall` statement, for example) and syntax for parallel array expressions. ABCL/1 uses syntactical constructs to differentiate the different message–passing modes it supports for invoking methods on other objects [118]. One mode is an *express mode*, which can interrupt the processing of a message sent in *ordinary mode* and even cancel the computation if desired. It also includes constructs for the parallel execution of a group of methods with an implicit barrier at the end. Act3, a actor–based language, includes syntax for accepting incoming requests, sending replies, and specifying replacement behaviours for an object [2, 1]. Some parallel languages, such as High Performance FORTRAN (HPF) [32] and OpenMP [26] mitigate the separation problem by inserting parallel directives as comments that are used only by special compilers. These directives can be added or removed with less effort. The only parallel programming language that does not suffer from this problem is P^3L (Pisa Parallel Programming Language) [8], a pattern–based programming language. A P^3L program consists of a set of named code fragments and a separate pattern description that indicates how the fragments and patterns are composed into a larger program.

Many parallel programming systems separate the parallel structure from the application code through indirection. In explicit message–passing systems such as DPnDP [97], Tracs [10], and Parsec [31], all messages are sent through channels via ports. The ports do not contain any reference to the process[1] that will actually receive the data, and so decouple the two communicating processes. Thus, a process may be freely interchanged with another that exchanges the same data. Parallel Architectural Skeletons (PAS) [41] take a slightly different approach. They create an additional process that serves as the fixed entry point to the processes that make up the pattern (a *representative* in PAS terminology). This fixed entry point allows a pattern to be replaced with another in a seamless manner. In PAS, the rest of the processes needed to form the pattern (the *back end*) only accept messages from either the representative or other peers in the same back end using a protocol dictated by the selected pattern. This approach was pioneered in Enterprise [89], where the representative was called a receptionist. Each protocol provides a set of message–passing primitives that are tailored to the specific needs of the protocol, so any change in pattern impacts the application code. Another separation technique, used by CODE[77], HeNCE [12], VPE [78], and Phred [13], is to use implicit process communication. Flagged variables in a process are implicitly sent to other processes at the end of its computation. Again, a process can be replaced with another that uses the same data. Not all programming systems maintain this separation, though. FrameWorks[2] [95] introduces keywords to distinguish parallel and sequential function calls, and requires the user to draw an external graphical description of the parallel structure, both of which must agree.

[1]We will use the term *process* to denote any parallel activity. This may be an actual process or a thread.

[2]The tool name should not be confused with generic object–oriented frameworks.

Enterprise requires the procedure calls in the application code to mirror the selected parallel structure drawn by the user. However, Enterprise does not require extra syntax to identify parallel function calls, so the parallel structure can be changed by simply changing a graphical representation, so long as the program code is consistent with the selected structure.

Other parallel programming tools are based on Cole's skeletons [24] or object–oriented frameworks [37, 54]. Skeletons are much like frameworks in that they provide a parameterized implementation of a particular parallel structure. However, where frameworks use inheritance in an object–oriented language to insert application–dependent code, skeletons are parameterized by a set of procedures that implement the application–specific responsibilities. Both skeletons and frameworks must be instantiated and the necessary procedures or objects must be assembled into the final application. Further, the application–specific code must adhere to the interface required by the structural code. These interfaces make it difficult to directly reuse the application code, as the interfaces will depend (at least partly) on the selected skeleton or framework.

2. Hierarchical Resolution of Parallelism (*Hierarchy*): A system should allow patterns to be composed hierarchically, refining the computation within a given pattern using another pattern. In general, a single parallel structure cannot be used to effectively parallelize all parts of a large computation.

   Since a user can generally place communication calls anywhere in application code, message–passing and thread libraries meet this characteristic. PUL and archetype libraries do not appear to have been composed, so it is unclear if this criteria is met. Similarly, most programming languages allow a programmer to annotate additional classes for parallel execution easily, allowing for hierarchical resolution of parallelism by allowing a parallel class to use other parallel classes as collaborators. The computation nodes in a $\mathrm{P}^3\mathrm{L}$ program can be replaced with another pattern in a hierarchical fashion. This refinement is possible because all $\mathrm{P}^3\mathrm{L}$ patterns have a single point of entry and exit, so substituting a pattern for a computational node is seamless. In both HPF and OpenMP, annotations can be incrementally added to procedures. However, HPF does not support nested parallelism. The OpenMP specification permits nesting, but individual implementations may not fully support it.

   Most parallel processing systems provide a mechanism for hierarchically specifying parallelism. The most common technique is identical to the approach taken by $\mathrm{P}^3\mathrm{L}$, replacing a simple computation node with another pattern or graph structure. Again, the interface for both computation nodes and larger pattern or graph structures is identical (usually a single entry and exit point), so the replacement is seamless. Some systems, such as HeNCE, do not support hierarchical resolution. Without hierarchical resolution, the user must draw one large graph for the structure of the complete program.

Programs built using skeletons or frameworks can experience composition problems for several reasons. The root cause of some of these problems is that frameworks and skeletons were created with the assumption that only one would be used in a given application. Some skeletons research is considering how skeletons can be safely nested. Framework composition is discussed in more detail in Section 2.3.2.

3. Mutually Independent Patterns (*Independence*): There should be no rules regarding how patterns can be composed. In other words, patterns should be context insensitive with respect to one another.

   Again, code libraries meet this concern because of their generality. For example, message–passing library primitives can be placed anywhere in application code, so long as the sends and receives are properly matched. Without evidence of composition, there is no way to know if PUL and archetype libraries are independent.

   The patterns supported by P^3L have no rules regarding their composition. Most other languages (save HPF, which does not permit nested parallelism) have no restrictions on the composition of parallel processes in their application programs.

   Among the pattern–based programming systems (DPnDP, Enterprise, FrameWorks, Parsec, Tracs, and PAS), only FrameWorks, the oldest system, does not exhibit independence of patterns. Specific combinations of structures could not be properly supported. Tracs patterns (or *architectural models*, in their terminology) are independent of one another, but require more work to compose. A Tracs architectural model is an arbitrary program structure drawn by the user, which does not have a fixed set of input and output ports. As a result, new ports may have to be added to an architectural model before it can be composed with another. HeNCE, VPE, and CODE represent programs as directed graphs, which can create any necessary structure. Phred uses a context insensitive graph grammar to construct the structure of a program.

   As noted in the discussion on *hierarchy*, frameworks and skeletons have outstanding issues with regard to composition. Again, these issues are discussed in more detail in Section 2.3.2.

4. Extendible Repertoire of Patterns (*Extendible*): A user should be able to incorporate new patterns into the tool. Ideally, these new patterns should be indistinguishable from the patterns originally supplied with the tool.

   Code libraries that provide general support for parallel programming, such as message–passing and thread libraries, can be used to create any desired parallel structure. The libraries for PUL and archetypes implement primitives for specific, higher–level parallel structures. It is difficult to use these libraries to implement new structures, but it is possible to create new parallel structures using the underlying run–time facilities (the CHIMP message passing system used in PUL, for instance).

Most parallel languages provide the means to implement new parallel structures. However, the programming model supported by the language can impose restrictions on the structures that can be supported efficiently. HPF and OpenMP have poor support for task parallelism, limiting the range of structures that can be efficiently implemented. The patterns in P^3L are fixed in the language, and cannot be extended.

CODE, VPE, and HeNCE satisfy this characteristic as they represent program structures using directed graphs, which allow programmers to create general structures. Phred programs are ultimately limited by the graph grammar rules that are used to create the application structure, but a programmer can create any structure within the rules. The rules, though, are fixed.

Of the pattern–based systems, only DPnDP, Tracs, and PAS provide any mechanism for creating new patterns. DPnDP provides a C++ framework that is used to create new patterns. The framework provides mechanisms for creating the process and communication structure of the pattern. However, only the structure of the new pattern can be specified; no behavioural parts of the pattern (such as synchronization or pattern–specific communication) can be addressed automatically. If a new pattern needs control messages (such as those needed between a replicated worker and its manager process, which is automatically handled by the pattern implementation supplied by DPnDP), the user code in the processes making up that pattern will need to exchange these messages explicitly. Tracs allows a programmer to create and save architectural models, which can then be loaded into the user interface for reuse. An architectural model is a graph with formal models for specifying the task and communication structure, akin to formal parameters in a procedure. This formal model names each part of the structure but does not supply any implementation. These names must then be bound to the specific models in the graph of the current application. A user can create a library of architectural models for reuse. One weakness, though, is that these models are static. A user cannot create a general architectural model for an `n`–stage pipeline or an `n`–way replicated process. To mitigate this problem, the Tracs interface provides operations to easily build repetitive structures. PAS uses C++ templates to associate the processes in a program with protocols that dictate the communication structure. New protocols can be added as subclasses of existing protocols. Unfortunately, it is unclear if the high–level specification language provided for PAS can be updated with these new protocols. However, the user can bypass the specification language and write programs in standard C++, where added protocols will be available.

Neither frameworks nor skeletons are generally considered to be extendible. In general, it is difficult to modify an existing framework or skeleton to support any type of computation beyond that for which the framework was designed. Like PUL and archetypes, it is possible to create new frameworks and skeletons. However, it may be difficult to incorporate new skeletons into tools that support skeleton–based development. For example, some skeleton systems use an external specification language to create programs [40, 39]. New skeletons can only be introduced if this language has a mechanism for integrating them.

5. Large Collection of Useful Patterns (*Utility*): The supplied patterns should cover a broad range of applications.

   Any systems that impose limits of the set of available parallel structures will have problems meeting this concern. Pattern–based systems have a limited number of supported patterns, which limit their applicability. For example, Enterprise, DPnDP, and P^3L do not support a pattern for mesh computations. The libraries for PUL and archetypes, as well as frameworks and skeletons, are targeted at specific parallel structures and problem domains. Some languages provide only a particular type of parallelism. For example, HPF and OpenMP are limited to data parallelism.

   The visual, graph–based tools and general–purpose libraries (HeNCE, CODE, VPE, PVM, MPI, pthreads) meet this characteristic by virtue of their generality. Other than the exceptions above, parallel programming languages are also general enough to be applied to many problem domains. However, this generality raises correctness concerns that are discussed later.

   In addition to the structured patterns, PAS provides a *compositional skeleton* in which the programmer can write programs with a high–level subset of a message–passing library. This skeleton serves as a catchall that allows any process structure to be created when none of the patterns suffice. Once again, the generality of this skeleton raises correctness concerns.

6. Open Systems (*Openness*): The programmer should be able to access low–level mechanisms, such as the underlying message passing system, in their applications. Otherwise, the programmer is limited to developing applications that can be expressed using the available patterns. Unfortunately, the implementation of the patterns may introduce undesirable overhead into the program or the structure of the pattern may not completely match that required by the application. With access to the low–level features of a system, these problems can be corrected. We

augment openness to include access to run–time libraries and generated code used by a tool, so that the programmer can modify the structure and tune the performance of an application.

Openness, in this sense, favours abstraction over transparency [60]. Transparent systems, advocated by distributed systems researchers, close off their designs from the end user. This is only reasonable when the implementation is very brittle and should not be changed or when the implementation is optimal and there is no need to change it. The main benefit to transparency is that system developers can assume that certain events or problems cannot occur because they control the complete implementation of the system. In contrast, abstraction encapsulates details so that they can be used in a transparent manner if desired, but allows access to the details if it becomes necessary. However, this access should be more than simply delivering source code. The details should be provided using abstractions that are easy to understand and use, yet still be flexible and sufficient to create a wide variety of alternative solutions. Finding these abstractions is more difficult than simply closing off the internals of a system.

Libraries can be considered open in the sense that any library that provides the correct interface can be used. A user can replace a library with a better implementation. It is also possible to augment a library by building new primitives by using the underlying facilities used in the implementation of the library itself. This will likely require access to the source code for the library. For instance, a programmer could add new PVM primitives by using the underlying sockets.

Most of the parallel programming systems mentioned to this point are closed. The abstractions in the programming model are transparent to the user. There is no opportunity to expose these details for tuning.

The only programming systems mentioned thus far that can be considered open are DPnDP and PAS. They permit the programmer to use the underlying message–passing system to send messages between any processes in a program. However, this openness does not extend to providing access to other run–time components, so there are no opportunities for performance tuning. The remaining programming systems hide their run–time systems by introducing special compilers or code generation, which transform the user–supplied code into a parallel application. This transformed code is hidden from the user. Parallel programming languages also use compilers to hide their run–time systems. PAS does not provide access to the underlying message–passing library, but does allow new protocols to be added to the system, possibly sacrificing the use of the specification language in the process.

Frameworks and skeletons are both considered closed. With a framework, it is possible to modify the structure by overriding structural code in subclasses. However, the primary benefit to using a framework is the reuse of this structure, so this is not normally done. In addition, the structural code may not be written so that it can be easily changed; the number and scope of the methods that have to be changed may be too large. In a parallel framework, the inheritance anomaly can exacerbate this problem even further [73].

2.1.2 Programming

This category deals with issues related to writing programs with a parallel programming system. In particular, the programming model supported by a system may force constraints on application code.

7. Program Correctness (*Correctness*): A good parallel programming system should provide some correctness guarantees. For instance, a system may guarantee that an application cannot deadlock. This characteristic can be extended to include reducing the possibility of programmer errors during application development as well. For instance, a tool can ensure that the application matches the desired parallel structure or that the correct type of data is sent and received between processes. However, no tool can prevent a user from introducing logic errors into program code or prevent the selection of an inappropriate parallel structure.

   General–purpose libraries offer little correctness guarantees. Both message passing and thread libraries require the programmer to correctly create the application structure, and do not provide any mechanisms to prevent synchronization or communication errors. The user of a message passing library also has to marshal and unmarshal the data in messages as well, which is a common source of errors. The libraries for PUL and archetypes provide library primitives that better support the selected parallel structure, but the user must still assemble the primitives into a complete application. This leaves room for error. For instance, library primitives for global communication or barrier synchronization need to be executed by each process in the application at the same time. If not, the program may deadlock.

   Parallel programming languages normally implement marshaling as part of the language, and can use compiler support to prevent any typing errors. However, except for P^3L, no language provides any mechanism for ensuring that a programmer has correctly implemented the desired parallel structure.

   All pattern–based programming systems offer some correctness guarantees. Unfortunately, most systems also leave some aspects of correctness to the application programmer, leaving open the possibility of user error. The patterns supplied with DPnDP implement all communication between processes that is specific to a pattern, such as control messages between a group of replicated workers and their manager process. However, any remaining communication, including marshaling and unmarshaling, must be implemented by the user. Further, the structure of a DPnDP program is a directed graph, where each node can be implemented using a design pattern. There is no mechanism for verifying that the graph represents the structure that was intended by the programmer. The protocols in PAS help ensure that the communication structure is correct, except for the composition skeleton which explicitly has no structure. Further, while PAS has some support for marshaling and unmarshaling objects, the user must still write some of the code. Enterprise uses a precompiler to ensure that the application code matches the desired parallel structure, generates correct communication code, and guarantees that an Enterprise program is deadlock–free. FrameWorks, Tracs, and Parsec require that all messages exchanged between processes be user–defined structures. These systems can verify that both sending and receiving ports are expecting the same structure. The user must fill in the fields of this structure, which is less error–prone than marshaling and unmarshaling.

Most graph–based parallel programming systems add type checking of messages between processes, but offer no structural correctness guarantees. In some systems, some of the local variables in a process are automatically forwarded to other processes, preventing marshaling and unmarshaling errors. Phred also has the ability to determine if the program is deterministic. Phred graphs describe both the application structure and access to shared repositories. The graph grammar supported by Phred allows these graphs to be analyzed for conflicting accesses to the data in the repositories. Unfortunately, this analysis was developed for the Phred grammar; it is not clear if this can be extended to more general graphs.

Skeletons and frameworks provide a correct structure that is used as the basis for an application. The possibility of error can be reduced by including all communication and synchronization that are specific to the structure. However, there is no way to prevent the user from adding application code that includes errors. In particular, the user could add parallel code that conflicts with the existing communication or synchronization policies used by the structural code.

8. Programming Language (*Language*): The system should use an existing, commonly–used programming language. Ideally, the syntax and semantics of the language should be unchanged so that users can directly reuse their existing sequential code in parallel applications. Further, the system can take advantage of expertise in an existing language.

   All libraries and frameworks exhibit this quality. Both are code provided to the user for building larger applications. They do not alter the programming language in any way. Some skeleton systems, such as [40, 39], use a separate specification language to instantiate the skeleton and provide application code. The application code in the specification language may be code fragments. The use of code fragments violates this characteristic, as the programmer is not writing normal code.

Parallel programming languages fail to meet this characteristic, of course. The degree of failure differs greatly, though. Orca, ABCL/1, and Act3 are completely new languages, which will have a steep learning curve for new users. In contrast, Mentat, ABC++, JavaParty, Titanium, and Spar are extensions to existing languages, adding new language features to support parallel programming. Because they are based on an existing sequential language, they can take partial advantage of existing code bases and expertise. ABC++, JavaParty, Titanium, and Spar primarily add new constructs to their base language, supporting features such as active objects, parallel array expressions, and parallel loops. Mentat includes new features as well, but also changes the semantics of method invocations to parallel objects to introduce *futures* [44]. A future is a variable whose value is the result of a parallel call that may not have completed. The future is *resolved* when its value is accessed, which requires the resolving process to wait for the parallel call to compute its value. Futures hide synchronization between parallel processes by making parallel calls appear as though they were simple sequential method invocations, but the differences in semantics can lead to subtle user errors that can affect program performance [99, 96]. This kind of subtle change to language semantics is precisely what this characteristic is concerned with. HPF and OpenMP make no changes to the base programming language, but use comments in the language to add annotations. These annotations are ignored by normal compilers, which generate a sequential program. However, the annotations are used to change the semantics of the underlying language, for adding parallel constructs and changing the visibility of certain variables (making them local to a given processor, for example).

Some pattern–based parallel programming systems share many of the failings already mentioned. Specifically, those systems that attempt to preserve a sequential coding style in the programming model have no alternative but to change existing language semantics. Enterprise, like Mentat, uses futures for delaying synchronization with parallel calls. FrameWorks uses a specification language for its programs, which also includes extra keywords in the application code. PAS also uses a specification language, but users can also write programs using standard C++ together with the class library provided by the system. The remaining systems, DPnDP, Tracs, and Parsec, use explicit communication, which does not require any language changes. Tracs has one exception in a feature it supports for reusing existing sequential code. The sequential procedure is automatically invoked in response to a message at the relevant process and the return value is automatically forwarded along any output ports, which is counter to the flow of data in normal programming.

9. Language Non–Intrusiveness (*Non–Intrusiveness*): The application code written by a programmer should not have to accommodate the programming model provided by the system. A system can satisfy the *language* characteristic yet still fail to meet this one. For instance, a message–passing library does not impose any changes to the programming language but may require the program to be restructured to accommodate the extra communication calls that need to be inserted by the user. The solution to both this problem and the *language* characteristic is to create a compiler that parallelizes sequential code. Unfortunately, current compiler technology does not do this well. In fact, this problem may never be solved since for some problems (such as sorting) the optimal parallel algorithm is not derived by parallelizing an existing sequential algorithm [94].

   All parallel systems suffer from this problem to some degree. General–purpose libraries require significant code changes. Message–passing systems require communication primitives to be inserted into application code. Both messages passing and thread libraries may require code to be reorganized, mainly to distribute the application code over the processes or threads used by the program. PUL and archetype libraries need similar changes.

   Parallel programming languages, particularly object–oriented languages, introduce their own limitations and restrictions on application code. Like libraries, these languages require that the code for an application be partitioned across parallel processes. In addition, ABC++, JavaParty, and Mentat restrict the use of object–oriented language features, such as inheritance, exceptions, and static variables, to accommodate a distributed memory model. Some of these languages restrict the argument types and will not pass pointer arguments or reference arguments. For example, ABC++ does not permit object parameters in method calls between parallel objects. The use of futures in Mentat also requires code changes to ensure that there is some work done between a parallel call and the resolution of any futures the call creates. If this is not done, there will be little to no parallelism. HPF and OpenMP programs need to be modified to create more parallelism by removing dependencies in loops. HPF programs in particular need to be reorganized as all HPF directives controlling a given parallel construct must appear in the same lexicographic scope. For example, a barrier in a parallel loop cannot be placed in a subroutine called from within a parallel loop.

Parallel programming systems exhibit many of the same problems as parallel languages and libraries. DPnDP, Tracs, Parsec, and PAS explicitly exchange messages. Enterprise uses futures, much like Mentat. Application code in CODE, HeNCE, VPE, and Phred needs to be split over the nodes of the structural graph.

Frameworks and skeletons are intrusive by their nature. Both provide a complete structure, including a fixed, limited number of points in the design for introducing application–specific code. The design must be accommodated in the creation of any application. The primary difficulty in using any framework is the time needed to learn this design and how to use the provided hook methods to implement a complete program [54].

2.1.3 User Satisfaction

A system must satisfy performance and usability constraints if it is to be accepted and used by programmers.

10. Execution Performance (*Performance*): It should be possible to achieve the best possible performance for a program, subject to the selection of the parallel patterns.

    General–purpose parallel libraries can provide the best performance. The use of library primitives can be optimized by a knowledgeable programmer to reduce communication and synchronization costs. The principal drawback to libraries with respect to performance is that they inhibit compiler optimizations. For instance, since the compiler does not understand the semantics of the library calls, code motion or reordering cannot be done safely. Higher–level libraries tend to have closed implementations. Closed systems generally have lower performance as they must be more general, which introduces overhead that cannot be removed.

    It is the closed nature of parallel languages that restricts their performance. There is no opportunity to remove any run–time overhead introduced in the abstractions that are provided by the language. Another problem is the restricted programming model of some languages. HPF and OpenMP only support data parallelism, so problems that need task parallelism will have poor performance. Similarly, ABC++, P^3L, Act3, and JavaParty concentrate on task parallelism and have performance problems on data parallel programs. Spar, Orca, ABCL/1, and Mentat (specifically, Braid) address both data and task parallelism. It is also possible to improve the performance of a parallel language with an optimizing compiler that works on the parallel code. However, building such a compiler is difficult, and few languages invest the effort. Many of the compilers for these languages translate the source to another intermediate language and compile the intermediate language with a freely available compiler. P^3L differs in that its compiler gathers information about the execution environment and uses this to generate an optimized *abstract machine* for executing the program on the given computer system.

Again, the closed nature of the run–time support for parallel programming systems is a limiting factor in performance. Performance studies for PAS, Enterprise, DPnDP, and CODE show slightly lower performance than equivalent message–passing programs.

Skeletons and frameworks also have performance issues because of the hidden structure. One method for improving performance is to limit the domain of the framework, which can allow optimizations that are not possible in more general frameworks. For example, Lea's fork/join framework implements parallel programs in which tasks are recursively subdivided into independent subtasks that are executed in parallel [63]. The solutions of the subtasks are combined into the final result. Since the framework assumes the subtasks are independent, they can be executed in any order by any processor, enabling idle processors to steal tasks. This improves load balancing and hence improves the performance of the application.

11. Support Tools (*Support*): The system should provide a complete set of tools for application development, including debugging and performance tuning steps. The abstractions provided by the support tools should mirror those in the programming model.

    Libraries do not generally provide support tools. However, popular libraries are sometimes the topic of support tools research, so tools can emerge. A brief survey of some tools for visualizing, monitoring, and debugging PVM programs can be found in [14]. A similar survey for MPI can be found in [28]. Similar profiling and debugging tools exist for pthreads and Java threads. The higher–level libraries, skeletons, and frameworks have no support tools, but rather must use standard profilers and debuggers.

    Parallel languages provide compilers but generally little else in terms of support tools. The exceptions are HPF and OpenMP. A survey of HPF tools can be found in [81]. Similar surveys can be found for OpenMP on the World Wide Web.

    Of the parallel programming systems, most provide a basic set of development and compilation tools. Enterprise has the most complete tool set, including not only development and compilation tools but also program visualization [64], debugging (including an execution replay mechanism) [50], and performance tuning tools [114]. They are integrated with the Enterprise programming abstractions, so the user does not need to consider new models when using these tools.

12. Tool Usability (*Usability*): The system should be easy to learn and use. This is a crucial aspect of any tool research and must be addressed before these systems are adopted by programmers. However, as shown by the dearth of usability studies to assess usability (with Enterprise [99, 96], Orca [112], and PIE [86] being notable exceptions), few researchers consider this issue. In their defense, usability studies are difficult and time consuming to conduct.

    Though they lack formal usability studies, the large number of applications built with PVM, MPI, HPF, and OpenMP can be taken as a partial testament to their usability. It is clear that these systems can be used by a wide variety of people to create parallel applications. However, message–passing and thread libraries are acknowledged as being the lowest–level of abstraction for parallel programming, so most parallel systems research is directed at providing higher–level models to simplify programming.

    The PIE study compared graduate experiences in PIE against experiences with MultiProcessor C (MPC) for two example programs [86]. The templates in PIE (or *implementation machines* (IMs) in their terminology) have two representations: an analytical representation, to help predict the performance of applications with the given machine, and a pragmatic representation, which provides a modifiable template implementing the selected machine. Their results suggest that the IM group found the machines easy to understand, and that they were able to create programs more quickly than the MPC subjects. Also, the average execution times favoured the IM group. A closer examination of the execution times for the four subjects in each group reveals that the best execution times for each group were close, with identical results for the first program and with the best IM solution 11.5% faster for the second program. However, the worst performers in the MPC group were much larger than the worst in the IM group, with execution times 37% and 21% larger respectively.

    The usability studies in [99, 96] compared Enterprise to message–passing libraries (PVM and NMP [70], a local message–passing library) and PAMS [56], a high–level tool adding parallel regions and parallel loops to C and FORTRAN. The experiment was conducted over several assignments in a graduate course on parallel programming. The identified strengths of the two high–level tools were:

    (a) Both systems prevented several common errors in writing parallel applications. The errors that users did make only affected program performance, not correctness.

    (b) Programmers were able to quickly create a working parallel program.

(c) In Enterprise, there was good tool support. In particular, the participants in the study made use of the replay tool that allowed them to visualize the execution of their program and identify performance problems.

The study also found several weaknesses in the high–level tools:

(a) While programmers could quickly create a working program, the closed programming model meant there was little they could do to improve performance. Many participants spent considerable time trying to circumvent the programming model, to no avail.

(b) The performance of programs written with the high–level tools was poorer than their message–passing counterparts. In combination with the lack of control over the high–level systems, this problem frustrated many programmers.

(c) In Enterprise, the use of futures introduced subtle changes to normal C semantics. Programmers who failed to account for these new semantics introduced performance problems in their code.

The study for Orca [112] is based on student experiences with the Cowichan problem set for evaluating parallel programming systems [110]. The Cowichan problems are a set of six modestly–sized programs that cover a broad range of application domains and parallel programming idioms. Six students spent between three and seven months developing sequential and parallel solutions to a given problem. The primary lessons from this study can be summarized as follows:

(a) Orca was easy to learn and use. A programmer must still consider synchronization and communication issues, but does not need to include parallel code. Instead, parallelism in Orca is based on explicitly forked processes, and communication and synchronization are based on shared data–objects with atomic functions.

(b) The programming model of a parallel programming language should be general. The Orca model had two limitations that required significant program changes to work around.

(c) Tools are an essential part of obtaining good performance. Since Orca provides a high level of abstraction and uses an advance run–time system, it is difficult to understand the performance behaviour of a program. In particular, a programmer needs to be able to determine if performance bottlenecks are the result of poor application code or poor decisions made by Orca's run–time system. For instance, Orca tries to automatically replicate objects over different processors if the object is read often but rarely changed. If the replication decision is incorrect, performance suffers. Unfortunately, performance problems created by this type of run–time decision cannot always be corrected by the programmer.

Another interesting study related to usability was one undertaken to assess the impact of design–pattern–based systems on parallel program complexity and maintainability. This study was based on several applications written with MPI and contrasted them with equivalent programs written with FrameWorks, Enterprise, and PAS [101, 100]. The study compared 30 code complexity metrics and three maintainability models, all based on static analysis of the application code from the applications. The study concluded that the programs written with the three pattern–based systems were less complex and more maintainable than the MPI equivalent. The study did not compare and contrast the different pattern–based tools, though. As well, the study did not consider any metrics for object–oriented programs as FrameWorks and Enterprise are both based on the C programming language.

Frameworks and skeletons have a high learning curve because of the structural details they encapsulate. The programmer must understand the flow of control through the objects contained in the framework in order to understand how and where to insert application–specific code. However, like most software tools, as a user becomes more familiar with a framework it becomes easier to create applications using it [54].

13. Application Portability (*Portability*): The system should allow applications to be ported to different architectures. The performance of a program may suffer on an inappropriate architecture, but the application should continue to run.

    Most systems provide some degree of portability by simply recompiling programs on the new architecture. Programs or programming systems built on message–passing libraries can generally be moved between different architectures in this way. However, the process–to–processor mapping may not be optimal, particularly on distributed memory machines with a distinct processor interconnect structure. Further, on shared memory machines, message–passing systems continue to communicate using expensive network messages rather than taking advantage of cheaper memory–based communication mechanisms. Thread libraries, on the other hand, cannot be easily ported to a distributed memory system without some additional software (such as a distributed memory system like TreadMarks [5]). Another approach, taken by Enterprise, is to support multiple communication managers, which abstract out the actual communication library or system, and can be configured at run–time to use either message–passing or shared memory. P^3L takes a similar approach, using the compiler and run–time system to create an abstract machine to execute a program. This machine is customized to its execution environment, based on the program structure and the underlying hardware. ABCL/1 has undergone several ports to new architectures, ABCL/onEM–4 [115] and ABCL/onAP1000 [117], to improve the implementation of the language for the different execution environments.

An important aspect of portability is the parallel algorithm used to solve the problem. The programmer must ensure that the algorithm works well on different architectures. The relative cost of communication and synchronization can vary greatly across different machines, which can cause the performance of a program to vary. Normally, a parallel program is optimized for its execution environment by explicitly mapping the parallelism and data onto the physical processors, limiting its portability. Portable parallel programs typically delegate this mapping to a run–time system. However, this automatic mapping is based on general strategies that may lower performance [61].

2.1.4 Summary

There are many parallel programming systems, tools, and languages, including some that are not mentioned here. From the above evaluation, it is clear that while some tools are better than others, all have some serious flaws. These flaws have prevented the high–level tools from moving into wide–spread real–world practice.

2.2 Design Pattern Research

2.2.1 Parallel Design Patterns

There are too many individual concurrent design patterns to enumerate here. The most notable source of concurrent design patterns comes from the ACE project [90], Lea [61], and the second volume in the Pattern–Oriented Software Architecture series [91]. These patterns concentrate on creating and managing concurrency, and not on the performance aspects of parallel programming. However, concurrency is a crucial aspect of parallel algorithms. Also, the structural patterns in these references are fine–grained, solving small and specific design problems. A typical concurrent program would need several of these patterns.

Common, larger–grained parallel structures, such as the work farm, pipeline, and mesh, have been known for some time now. Details can be found in the introductory parallel design and algorithm literature [33, 22].

There has also been some research in generating code for design patterns. Budinsky *et al.* [21] have created a web–based tool that generates code for the patterns in [37]. This tool also allows the user to select from the alternative pattern implementations found in the pattern documentation and generate appropriate code. The complete set of classes for the pattern are returned to the user, who then adds application–specific functionality.

Other patterns research has focused on using patterns to verify that a program implementation adheres to its design [92]. This is done by ensuring that the interactions between the different elements of each pattern are correct. Any interaction between objects that is not part of the pattern represents a place where the implementation diverged from the original design, either during initial development or subsequent maintenance. One result of the work in this area is *Pattern–Lint*, a tool that uses a combination of static and dynamic information to gather the collaboration structure of a program. This tool was used to fix several errors introduced in several subsystems in an object–oriented operating system. This work also suggests more proactive approaches to verifying program design. First, they suggest generating code for the design patterns to ensure that they are correct initially. Second, they advocate inserting run–time assertions to prevent the pattern implementation from diverging from its correct structure.

2.2.2 Productivity Benefits of Using Design Patterns

Advocates of design patterns have provided anecdotal evidence of their benefits. These benefits include improved program design and improved communication between designers, developers, and maintainers of a software system. These improvements are partially attributed to the vocabulary introduced by the use of patterns; it is possible to exchange design information at a much higher level than individual classes. However, much like the lack of usability studies in parallel programming systems, there was no initial attempt to determine if these benefits truly existed and, if so, to determine more precisely how these benefits were accrued.

To answer these questions, some researchers conducted a series of controlled experiments to ascertain the usefulness of design patterns [84, 85]. These experiments were designed to determine if documenting the design patterns used in a program improved its maintainability and to determine if programs that used design patterns were more maintainable than equivalent programs that did not.

The conclusions of the experiments are mixed. Explicitly documenting the patterns in a program improved its maintainability. Maintenance tasks that involved the patterns were performed faster and with less errors for those programs that included pattern documentation. However, the experiment also revealed that the indiscriminate use of design patterns where a simpler solution exists is sometimes detrimental. This effect may by a function of the specific pattern, as some patterns are acknowledged to be more complex than others. The use of a design pattern must be weighed against its simpler alternatives. However, when in doubt, a design pattern is the preferred alternative. A pattern will increase the flexibility of the solution and may be helpful to maintainers in non–obvious ways.

2.2.3 Parallel Pattern Languages

A design pattern is sometimes defined as a solution to a problem in a context. One essential aspect of patterns is that they provide a common vocabulary for communicating design information.

As the number of patterns grows, the application of some patterns naturally leads to others. As these relationships between patterns grow, a structure may emerge and form a cohesive set of rules for applying the patterns to solve larger, more complex problems. These rules dictate the subset of patterns that should be used and the order in which they should be applied to solve a design problem. They provide a means of using a set of design patterns to deal with problems that cannot be handled by any one pattern. This collection of patterns and rules forms a *pattern language*.

A concrete example of a pattern language can be found in [72]. This pattern language is targeted at finding the most appropriate parallel structure for a given problem. The rules guide the designer through a set of analysis patterns that are used to find the concurrency and identify how it should be decomposed. At the end of this process, the analysis leads to a parallel structure for the problem.

This design stage is absent in tools research. Tool developers assume that the programmer is familiar with the design of parallel programs and concentrate on support for development stages.

2.3 Framework Research

Framework research is not limited to creating new frameworks and using them to build applications. As frameworks become more prevalent, new issues have become apparent.

One obvious issue is the documentation of frameworks. As already noted, in order to build an application, the programmer must understand enough about the flow of control through the objects to understand how and where to insert application code. Without adequate documentation, this task could prove impossible. Section 2.3.1 discusses work in framework documentation.

Another issue that was raised in Section 2.1 was the composition of frameworks. Many applications span several problem domains, each with a useful framework. Composing these frameworks may not be trivial, though. Section 2.3.2 looks at some of the problems that can occur and their solutions.

Finally, another problem that arises in frameworks is how they are instantiated. Normally, this instantiation requires the user to understand many architectural and implementation details of the framework. The number of classes and objects that need to be created in some frameworks can be large, which places a significant burden on the user. Section 2.3.3 looks at a language–based solution to this problem.

Note that these are not the only problems related to object–oriented frameworks that have been identified. Others include implicit architecture and cross–framework dependencies [16], which are not discussed here.

2.3.1 Documenting Frameworks

In documenting frameworks, the objective is to provide sufficient detail so that the programmer can determine how to use the framework to implement an application. There have been two basic approaches: documenting how the framework is designed and documenting how to use the framework to accomplish a specific goal.

A number of methods have emerged to document the design of a framework. One popular method is to document the framework in terms of the design patterns that were used to implement it [53]. One of the improvements that a design pattern can have on a software system is the introduction of flexibility in the design. Each pattern allows some aspect of the design to evolve over time, independently of other parts of the design. Often, understanding the design patterns used in some aspect of a framework can lead to an understanding of how to specialize that aspect for a given application. The Template Method and Factory Method patterns [37] require that a class be subclassed and that certain primitive methods be overridden with new implementations. The Observer pattern [37] requires that observing objects register to be notified whenever a subject object is changed, so that the observers can be kept up–to–date. It also requires that a subject object must provide the ability to register observers and must ensure that observers are notified of any state changes that they are interested in. It has been noted that patterns describe not only the structure of the design of software, but also its architecture and the reasoning behind it [11].

Rather than describing the design of the framework, *hooks* are intended to show how to use a framework to to accomplish specific tasks that are part of creating a complete application [35, 36]. Each hook shows which parts of the framework must be changed or augmented to incorporate application–specific functionality into a framework, including any constraints that must be observed. A hook may also include the use of other hooks if the needed changes are large enough. The developer of a framework supplies a complete list of hooks showing how to specialize it. Hooks can be characterized using two different axes: method of adaption and level of support. Level of support, in particular, has three categories:

1. Option hooks. This level provides the most support. For some part of the framework, the user has a choice of a number of different pre–built components. A new component can be selected, without requiring extensive knowledge of the framework design.

2. Supported pattern hooks. The framework developer has defined an interface for fulfilling some part of the requirements of the framework, but the implementation is application–dependent and must be filled in by

the user. This can be as simple as supplying parameters to some of the framework classes or as difficult as creating new subclasses and overriding methods. The important characteristic of this category of hook is that the framework was designed with the intent that these changes be made. The user will need to know more about the framework to use these hooks than is needed to use option hooks, but will not need a deep understanding of the overall design.

3. Open–ended hooks. These hooks have the least amount of support. These hooks show the user how to make changes that the framework developer choose not to support or could not support. These hooks can involve any type of changes to any of the classes and methods in the framework, including changes to the structural classes. If an open–ended hook is used frequently, this may be an indication that the framework should be changed to make it an option hook instead.

2.3.2 Composing Frameworks

Frameworks were initially created with the assumption that only one would be used in a particular program. This assumption has proved to be incorrect. As programs grow larger in both size and scope, multiple frameworks are needed to cover the complete problem domain. Unfortunately, composing these frameworks into one large application can be difficult. This section briefly examines the problems that have been identified, the underlying causes of these problems, and potential solutions [75, 76].

In total, five primary composition problems have been identified. These are:

1. Composition of Framework Control. Most frameworks define the flow of control through the program. For example, GUI frameworks take over the flow of control by going into an event loop and invoking user code only in response to events in the user interface. Composing two such frameworks together requires that the two separate flows of control be merged.

2. Composition with Legacy Systems. Many frameworks rely on subclassing framework classes to introduce application–specific code and to enforce the interfaces between framework components. Legacy systems will not be subclasses of framework classes and will probably not support the correct interface needed by the framework. They will need to be adapted for the framework.

3. Framework Gap. The composed frameworks may not cover the application requirements.

4. Composition Overlap of Framework Entities. Two frameworks may each have a representation of the same entity, implemented from its own per-

spective. These representations need to be composed as well. However, the state in the representations may overlap, creating dependencies between different state elements that must be maintained. For example, one framework could treat a sensor as a fire alarm that has either tripped or not. Another framework might be using the same sensor to measure temperature and pressure. When combined, there is a dependency between temperature and the status of the fire alarm. Any changes to the temperature require that the status of the fire alarm also be updated.

5. Composition of Entity Functionality. This problem occurs when the functionality of an entity requires the composition of parts of functionality from different frameworks. Changes to one part of the entity must be propagated to all relevant framework functionality. For example, an entity in an application framework may also need to be displayed on a user interface and need to be persistent. A change in the entity will not be automatically reflected in the other frameworks.

The four underlying problems at the root of each of these composition problems have also been identified. These are:

1. Framework Cohesion. A class in a framework contains two types of functionality, domain–specific behaviour to implement the necessary entities for the problem domain, and interaction behaviour to exchange information between framework components. The interaction behaviour is cohesive in that it establishes how the framework classes operate together. It is not a function of the problem being solved, but is rather a function of the framework structure. To replace a class in a framework, it is necessary to properly implement both the domain–specific behaviour and the interaction behaviour.

2. Domain Coverage of the Framework. Since there is no standard definition of the complete domain for most problems, there is no way to ensure that either the domain is covered or that the domain will not overlap with other domains. Thus, frameworks built from these inexact specifications can experience either framework gap or overlap.

3. Design Intention of the Framework Designer. The design intentions for the framework are often not detailed enough to decide if it will be possible to compose different frameworks. Some frameworks exhibit properties that make them amenable to composition while others do not.

4. Lack of Access to the Source Code. Some of the changes needed to address the composition problems are best addressed by modifying the framework source code. If source code is not available, the user will have to wrap framework objects inside other objects to change the behaviour of the framework, at the expense of performance. Note that this effect can happen if the source code for the framework is complex enough that the user decides against trying to understand and change it.

	Composition of frame–work control	Composition with legacy components	Framework gap	Overlap of entities	Composition of entity functionality
Cohesive behaviour	Complicat–ing factor	Primary cause	–	Complicat–ing factor	Complicat–ing factor
Domain Coverage	–	–	Primary cause	Primary cause	–
Design Intention	Primary cause	Complicat–ing factor	–	Complicat–ing factor	Primary cause
No source code access	Complicat–ing factor	Complicat–ing factor	Complicat–ing factor	Complicat–ing factor	Complicat–ing factor

Table 2.1: Framework composition problems and their causes [75].

Table 2.1 summarizes the causes of each of the composition problems.

Despite all of the problems with framework composition, there are a number of potential solutions to each of the five problems. Some of these solutions are:

1. Concurrency. For the composition of framework control, it may be possible to use a different thread for each framework rather than merging the control loops. This only works if the frameworks are independent (*i.e.* no framework needs to be notified of any events in another framework). However, this solution will require synchronization in any objects that may be accessed by multiple sources.

2. Wrapping. The user can wrap a framework object inside another object. This wrapper adds functionality to the object it replaces. For example, when dealing with composition of framework control, the wrapper could be used to propagate events between different frameworks. When dealing with framework gap, the wrapped object can extend the framework object with new functionality.

3. Adapters. Composing legacy code is an example of a problem for the Adapter design pattern [37]. This pattern allows objects with different interfaces to work together by translating between the two interfaces.

4. Aggregation. One option for combining framework representations is to create a new aggregate class that holds an instance of each representation. This aggregate representation is used by all of the combined frameworks. One drawback to this solution is that the aggregate class must support the interface and interaction behaviour needed for each framework. Another drawback is that the source code for each of the composed frameworks may have to be modified to use the aggregate instead of its own representation. This solution can be used to address the overlap of entities and the composition of entity functionality.

5. Inheritance. It may be possible to compose the entities from different frameworks using an inheritance relationship. Entities with disjoint and independent representations of an entity can be composed using multiple inheritance. This solution can address framework overlap and composition of entity functionality. Another possible use of inheritance is to combine subclassing and aggregation. For two frameworks with different representations of the same entity, a subclass of each representation is created that contains a reference to an instance of the other. Any changes to one representation can now be propagated. This can be applied to framework overlap problems.

6. Observers. When handling problems with the composition of entity functionality, the Observer design pattern [37] can be used to keep the different framework entities up–to–date. The observers register on events in one framework and update the others to keep the composition consistent. A variation of this solution is to use an event mechanism to keep different entities informed of any changes. To allow the different frameworks to enforce their constraints, each framework can publish an event informing the other frameworks of a proposed change before it takes place. Another framework can veto this change if it would violate a constraint [29].

The Catalysis approach supports component development with composable framework models [29]. A framework model is similar to the architectural models in Tracs. The model defines an abstract specification of the framework structure, providing generic types for the different components and generic relationships for expressing the structure. These models can also include constraint information to ensure they are used properly. To apply these models, the generic types and relationships are mapped onto the types and relationships in existing code. To distinguish the part a class plays in a framework, we say that a class plays a specific *role* in a given framework. Multiple frameworks can be composed by mapping roles from different frameworks onto the same class.

An alternative composition mechanism with Catalysis is to use *plug–points*. These plug–points are objects or attributes shared by multiple frameworks that glue them together. A plug–point can act as an adapter (described above), invoked from one framework and in turn invoking methods on another. The plug–point can also connect frameworks using an event mechanism, relaying information between frameworks indirectly.

Despite these problems, frameworks have been successfully composed in practice [75]. While we do not attempt to solve any of these problems in this dissertation, we should be aware of the potential problems. This way, our frameworks can attempt to avoid the problems that will limit composability.

2.3.3 Instantiating Frameworks

The last step to using a framework is to instantiate it, creating all of the objects that are needed for an application and composing these objects to create the necessary collaborations. This is not necessarily an easy problem, as there can be many objects that need to be created and composed. In addition, it typically falls to the user to create all of the necessary objects, which requires knowledge about the entire structure of the framework.

To alleviate this burden, some researchers are investigating language support for object–oriented frameworks. On such example is CORRELATE, a class–based concurrent object–oriented language based on active objects [74]. The run–time system for CORRELATE is a framework that implements a variation of the Active Object design pattern [59]. This framework provides an activation queue and scheduler to process the pending method invocations. To use the framework, though, a separate class must be created for each method that can be invoked on the active object. For a class with many public methods, creating all of the necessary classes and instantiating the complete framework is a considerable amount of work. Instead, the CORRELATE compiler introduces extra syntax to indicate both active objects and invocations on active objects. At compile time, this syntax is used to create the necessary classes and instantiate the complete framework. To prevent the framework from becoming closed, CORRELATE exposes the important abstractions (like the activation queue and scheduler) using a metaobject protocol.

2.4 Object–Oriented Modeling Languages

Tools for object–oriented programming in other domains have also used code generation and layered approaches. This section discusses a small selection of such tools from the sequential and real–time systems domain.

2.4.1 Generating Code from UML

UML (Unified Modeling Language) provides a standardized graphical representation for the design of object–oriented programs [15]. It is the amalgamation of several successful methodologies from early work in object–oriented analysis and design.

As with patterns, one of the traditional problems with UML diagrams is that they are design constructs. Implementation is a separate step in which the model is translated into constructs in a chosen programming language. As with patterns, this is a time–consuming process. Furthermore, it is possible for the program to diverge from the design model, which reduces its benefits. In response to these problems, researchers have developed several ways of using the UML diagram in a more direct manner during program implementation. Two examples of this work are code generation from a UML model and a UML virtual machine that directly executes the model.

One example of generating Java code from a UML representation can be found in [46]. This work is noteworthy because it places few constraints on the models for which code can be generated. One problem with UML is that it supports general object–oriented models without regard to the features of a particular programming language. Typically, programmers are forced to confine themselves to a subset of the modeling language that can be easily translated into the chosen programming language. For example, the model for a Java application would normally have to avoid multiple inheritance. However, the code generator in [46] is able to map a larger range of UML design elements to Java code, providing the designer with more freedom to model their application without the constraints imposed by Java.

With a code generation system, the application development cycle consists of changing the model, regenerating code, then recompiling and running the new program. For large systems, this cycle can take hours to complete. The UML virtual machine avoids this cycle by interpreting and executing UML models [87]. By avoiding this cycle, the programmer can quickly experiment with alternative models. However, once a design has been selected, production systems are created using code generation to avoid the overhead of the interpreter.

The UML virtual machine uses a layered approach in its meta–model, which is used to represent the logical architecture of the model. Each level of the meta–model provides a description of the previous layer. These levels, from lowest to highest, are:

M0 This level represents the objects in a currently executing program. These objects are also called user objects or domain objects.

M1 This level consists of objects for the model of the currently running program. These objects are the user classes or domain classes in the program. These objects describe the M0–level objects.

M2 This level contain objects representing the modeling language itself, which is UML in this case. This is also called the meta–model level.

M3 This level consists of objects that are used to describe the language in which the modeling language is represented. It is also called the meta–meta–model.

These levels are illustrated in Figure 2.1.

The benefits of these tools, with regards to the 13 characteristics, are the same as those for visual parallel programming systems in Section 2.1. The primary benefit is *separation*. The structure of the program is captured diagrammatically rather than being part of the user code. In addition, both systems represent associations between design elements as objects, further increasing the separation between them. These associations are similar to the ports and channels used to separate different parallel entities. However, in

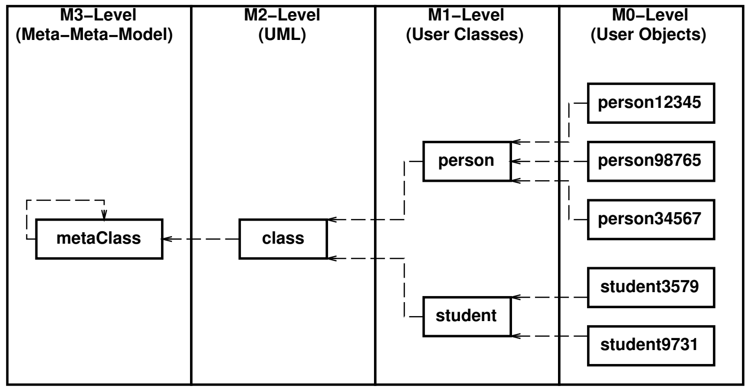


Figure 2.1: The four–level logical architecture of a UML–based system. The dashed arrows indicate an instance–of relationship.

terms of pattern support, these two UML tools share the same drawbacks as their visual parallel counterparts.

2.4.2 The ROOM Modeling Language

The ROOM[3] modeling language is an object–oriented approach for designing and implementing real–time systems [93]. The language is based on executable models, much like the UML virtual machine.

The conceptual framework of ROOM is split into two modeling paradigms: *abstraction levels* and *modeling dimensions*. These two paradigms classify the modeling concepts and help the programmer navigate through the design space.

The abstraction levels separate the modeling concepts for different scopes of a software system. The different layers both complement and constrain each other. Each layer deals with its own individual concerns, but lower layers must still be consistent with the model created at the higher layers.

There are two abstraction levels in ROOM: the *Schematic Level* and the *Detail Level*. The Schematic Level is used for issues such as concurrency and distribution, as well as other high–level behavioural modeling. Concurrent constructs at this level include ports, the bindings between ports, and actors (a concurrent object whose internal state and behaviour are represented by an extended finite state machine called a ROOMchart). The Detail Level is concerned with the modeling and behaviour of the non–concurrent parts of a system, such as data objects (such as strings, numbers, or records). These are implemented using traditional programming languages.

For specific problem domains, it is possible to add more modeling layers to the abstraction levels, either above the Schematic Level or below the Detail

[3]Real–Time Object–Oriented Modeling.

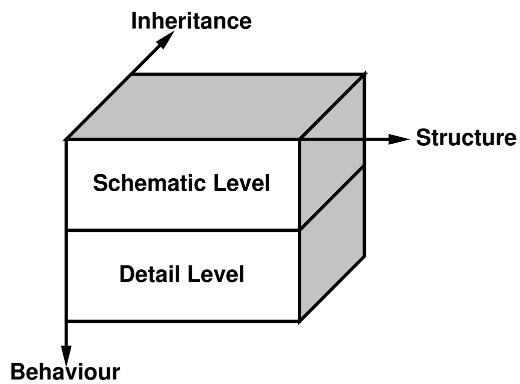


Figure 2.2: The conceptual framework of ROOM, taken from [93].

Level. For example, languages for modeling hardware systems can be added below the Detail Level if necessary.

The modeling dimensions paradigm partitions modeling concerns based on structure and behaviour. The structure defines the components in a system and their relationships. The behaviour defines how these components function. Since ROOM is an object–oriented modeling language, inheritance must also be addressed. Although inheritance seems to be part of the structure of a system, both the structure and behaviour of a system in ROOM are considered at the object level. Inheritance expresses relationships between classes. Further, it has been noted that it takes considerable effort to create well–formed inheritance hierarchies. Thus, the modeling dimensions paradigm considers structure, behaviour, and inheritance as separate concerns.

The complete conceptual framework of ROOM, showing the combination of both paradigms, is given in Figure 2.2. Each concern in the modeling dimensions paradigm must be considered by the two abstraction levels, although the form differs. For example, the structure of a software system at the Schematic Level is modeled in terms of actors connected by bound ports. At the Detail Level, the behaviour of different components is given as C++ or Java code. Each of the two levels deal with the structure of the system but at different scopes. Similarly, the behaviour and inheritance dimensions are considered by both of the abstraction levels in different ways.

It is important to note that these layers and abstractions are used to partition different modeling concerns. They are not used as a means of building up high–level services based on a lower layer. Since the model of each layer represents the same system, each layer is interrelated and must be consistent. This consistency is enforced by tools that support the ROOM modeling language. Further, these tools can help a developer navigate through the modeling activities that are necessary to create a software system.

2.5 Summary

This chapter used the 13 characteristics of ideal parallel programming systems to critically evaluate a cross–section of related research in parallel programming systems, languages, and frameworks. Each of these systems falls short in one or more of the categories. As we will see later in this dissertation, careful consideration of these 13 characteristics and the failings of existing systems guided the creation of both the PDP process and the CO_2P_3S parallel programming system. Other work in design patterns, pattern languages, and frameworks was also discussed. Finally, a selection of modeling languages, which also take advantage of code generation and layered approaches, were also presented.

Chapter 3

The Parallel Design Patterns Process

This chapter describes the Parallel Design Patterns (PDP) process, a new pattern–based approach for building object–oriented parallel programs. The process is novel in its approach to openness [96], which permits a user to access low–level facilities in a parallel application. The process supports three layers of user–accessible abstraction, where each layer provides successively lower–level control over application development and tuning. In contrast, many parallel programming systems are closed and do not permit the user to access or tune system–specific code. Those systems that do provide openness typically strip away all abstractions immediately, leaving the user to contend with a large number of implementation details that must be considered.

Over the course of this chapter, we will use the terms *structurally correct parallel program* and *semantically correct parallel program*. Within the context of this work, a structurally correct program refers to a parallel framework that can be used to write a program with the following properties:

- It correctly creates all of the necessary parallelism based on framework arguments.
- It correctly distributes data across the processes or threads used to execute the program.
- It provides a correct synchronization structure. There should be no race conditions or unprotected accesses to shared resources or data encapsulated by the program structure. Such data could include references to objects that represent the state of a parallel computation. This synchronization will be pessimistic, assuming the worst case.
- It provides a correct communication structure. Each process or thread will send the correct data to the proper collaborator. In a shared memory environment, there is no explicit communication so this characteristic means that shared data is used correctly. For example, each process or

thread will correctly access only the data that are assigned to it. Again, this communication may be pessimistic, sending too much data on some occasions.

- Neither the synchronization nor the communication structure will cause the program to deadlock.
- It will terminate properly and can shut down all of the parallelism. In many cases, the framework is given a mechanism to detect completion of its computation and will shut itself down. In other cases, where the amount of work in a problem cannot be determined beforehand, the framework provides the necessary means to release processes and threads when the programmer indicates that the computation is over. However, the parallelism should not be shut down until the work that currently being processed has finished.

This definition does not take into account the user application code that is inserted into the framework. It is possible for the programmer to write hook method code that will cause the program written with the framework to fail to meet some of these characteristics. For example, the data objects encapsulated by the framework may include objects to which the programmer has added hook method code. It is possible for these hook methods to include unprotected accesses to shared data (by using a static variable, for example). The above guarantee ensures that the structural code will properly access the object references, but cannot guarantee that those objects do not in turn incorrectly access shared data themselves. Unfortunately, such errors can be introduced by library code, which can make them difficult to locate and correct.

A semantically correct parallel program is a complete program, consisting of the structural code and user code, that has the above properties and solves the desired problem. The definition of semantically correct considers correctness but not performance. Where appropriate, performance is addressed separately. Note that a program is structurally correct but not semantically correct if the selected parallel structure cannot be used to solve the problem (either because of an incorrect parameter value or because the structure is wrong for the problem).

Section 3.1 presents a brief overview of the PDP process, presenting the three layers of abstraction and outlining the steps involved from the perspective of a user who is building a parallel application. To make the process more concrete, Section 3.2 illustrates these steps by walking through the development of an example application using the CO_2P_3S parallel programming system. CO_2P_3S is one (of many possible) tool for implementing the PDP process, and is described in more detail in Chapter 4. Section 3.3 outlines the benefits that the PDP process offers parallel programmers.

In addition to considering how a programmer might use the PDP process to write parallel applications, we must consider how tool developers might go about developing and supporting the abstractions in the process. Section 3.4 discusses these issues.

3.1 Overview of the PDP Process

The complete PDP process consists of five steps that support three layers of abstraction. The higher–level abstractions create structurally correct parallel programs and enforce correctness using encapsulation. Lower–level abstractions gradually expose implementation details and give users the opportunity to modify this structure and tune the performance of their applications.

The abstractions derived by the PDP process are:

The Patterns Layer This layer promotes the rapid development of structurally correct parallel programs. The user selects a *parallel design pattern template* from a palette of supported templates. This template represents a family of frameworks for a given design pattern. The user can select the member of this family that is best suited for an application by specifying application–dependent template parameters. Once the parameters have been specified, the template is then used to generate object–oriented framework code implementing the pattern indicated by the template. This code consists of abstract classes that correctly implement the parallel structure of the pattern template together with concrete subclasses that are used to insert application–dependent sequential code (*i.e.*, a structurally correct parallel program). Structural correctness is ensured by restricting access to the abstract structural classes; the user can only implement the hook methods in the concrete classes, which do not require any parallel code to work properly. A complete application consists of either a single framework or several frameworks composed together.

The Intermediate Code Layer This layer provides a high–level, object–oriented, explicitly-parallel programming language, a superset of an existing OO language[1]. The abstract structural classes are implemented using this language and are made available to the user. The user can modify the generated structure, write new application code, or tune the structure.

The Native Code Layer At this layer, the intermediate language is transformed into code for a native object-oriented language (such as Java or C++). This code provides all libraries used to implement the intermediate code from the previous layer. The user can tailor the libraries for the application requirements or for the execution environment.

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

To place other research work into perspective, we can consider them in terms of the above abstractions. Many systems are closed, providing an equivalent to one of the above three layers (typically the Intermediate Code Layer) and nothing more. The programming model provided by these systems provides an abstraction for creating parallel applications but hides the implementation details from the user. There is no opportunity to expose these details for tuning. DPnDP, the only system that provides any degree of openness, does not open up the implementation of its supported patterns. Rather, the tool permits the user to pass messages between any processes using the underlying message–passing system, which is a weaker version of the Native Code Layer. There is no possibility of tuning the run–time support for any of the patterns in DPnDP. In contrast, the above three layers identify a set of abstractions that open up the implementation of a parallel programming system in a controlled manner. The Intermediate Code Layer exposes these details using a high–level view that is conducive to making structural changes to the generated framework code. By providing a high–level view, novice parallel programmers may be comfortable enough with this layer to try tuning their applications. More experienced users can use this layer for high–level changes or they can use the Native Code Layer to examine and refine the low–level run–time support.

The PDP process has five steps to develop a complete parallel application. The first three steps are required and the last two steps are optional:

1. Identify the parallel design patterns that are required to parallelize the application and select the corresponding design pattern templates.

2. Supply the application–specific parameters for the selected templates and generate the framework code.

3. Implement the application–specific sequential hook methods in the generated frameworks, as well as any other application code. The Patterns Layer is now complete. Check that the result is a semantically correct parallel program. If not, return to Step 1.

4. If the performance of the parallel application is not satisfactory, use the facilities of the Intermediate Code Layer to examine the structural framework code. This code contains high–level synchronization and communication primitives. Any primitives that are not needed for the particular application can be removed. The use of the remaining primitives can be optimized (for instance, by changing their location in the framework).

5. If the performance of the parallel application is still not satisfactory, specialize the implementation of the primitives at the Native Code Layer. The new implementation can take into account the characteristics of both the application and the target execution environment (machine architecture, available libraries, *etc.*).

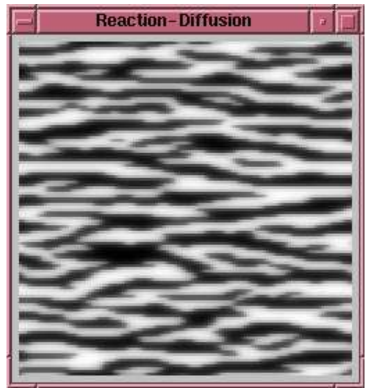


Figure 3.1: An example texture generated by the reaction–diffusion texture generation program.

3.2 A Detailed Look at the PDP Process

To make the discussion on the PDP process more concrete, this section examines each step of the process in more detail by examining the development of a simple parallel application using the CO_2P_3S parallel programming system. The details of CO_2P_3S are presented in Chapter 4. For now, it is sufficient to note that CO_2P_3S supports a Two–Dimensional Mesh pattern (referred to as a Mesh pattern from this point on) and that the tool generates multi–threaded Java frameworks for shared memory multiprocessors.

The example application is a reaction–diffusion texture generator from computer graphics. This problem simulates the reaction and diffusion of two chemicals called *morphogens* over a two–dimensional surface [113]. Starting with random concentrations of each morphogen and the correct reaction and diffusion parameters, the result of the simulation is a texture that approximates zebra stripes, as shown in Figure 3.1. Note that this texture can be seamlessly tiled on a larger surface. Computationally, the simulation is similar to simultaneously solving two interacting LaPlace equations and is solved using convolution. Each element on the surface computes its new values based on its current concentration of both morphogens (reaction) and the concentrations of its immediate neighbours (diffusion). To generate correct results, the problem uses Jacobi iteration, where the new value is computed based on the results of the previous iteration. The algorithm iterates until the change in morphogen concentration at each point on the surface falls below a threshold.

3.2.1 Selecting a Design Pattern

A parallel design pattern encapsulates a strategy for solving a problem using a familiar parallel structure. For a given problem, the user must determine which combination of patterns will lead to the best solution. Note that there may be many pattern combinations that work for an application. In a pattern–based tool, the options may be limited to the patterns supported by the system[2].

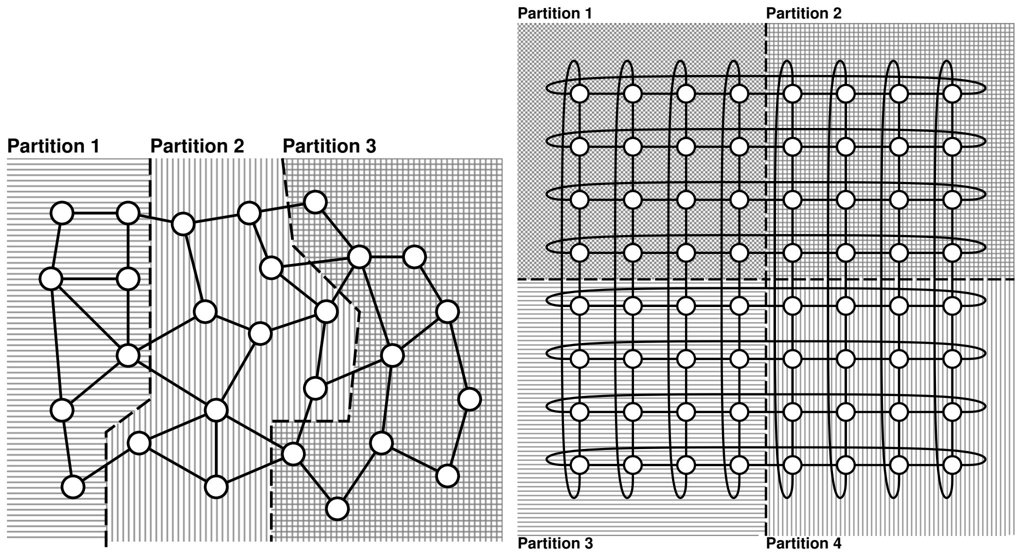
This selection problem, the first step of the PDP process, is not addressed by this work. This is a difficult problem that requires the user to identify the potential parallelism in the problem and understand how to exploit it. Obviously the user's understanding of the available parallelism is a limiting factor in selecting the parallel patterns. The user might select an inappropriate pattern for the problem. However, there are several methodologies for selecting the most appropriate parallel structure for an application [22, 33]. A parallel pattern language, such as the one described in Section 2.2.3, can also be used.

To keep the discussion simple, the reaction–diffusion example uses a single instance of the Mesh parallel design pattern. Other design pattern combinations can be used to solve this problem. The Mesh pattern naturally corresponds to the structure of this problem. More often, a parallel program will consist of several patterns, each responsible for parallelizing a different part of the overall computation. Some applications of this kind are presented in Section 5.

The Mesh parallel design pattern supports iterative computations over a set of connected data. More precisely, the input data is a graph consisting of nodes (or elements) and a set of edges connecting each node to a set of neighbours. Each node computes a new value based on a combination of its value and the values of some neighbourhood around it. To parallelize this kind of computation, the graph is split into a set of partitions, each of which is assigned to a process. The number of partitions (and hence processes) is usually the number of processors that are to be used to solve the problem. An example of a general mesh is shown in Figure 3.2(a). This solution requires communication between the partitions to exchange the boundaries, which are needed to compute new values for elements on the edge of a partition. The primary difficulty in applying this pattern is to simultaneously minimize the cost of the boundary exchange and balance the amount of computation for each partition. An example of a parallel programming tool that supports this type of computation is the PUL project. PUL includes a separate utility for this partitioning task [104], with the results fed into the library supporting mesh computations [105, 20].

The Mesh pattern can be applied to the reaction–diffusion problem in a straightforward manner. Each node of the input graph represents a region of the surface, and holds the concentration values of both morphogens. These elements are arranged in a regular, two–dimensional array with each node connected to its immediate neighbours. To generate a texture that can be seamlessly tiled, the elements on the edges of the data are linked to elements on the opposite edge in a toroidal fashion. This graph can be split into regular, rectangular partitions. The result is shown in Figure 3.2(b).

[2] In some tools, it may be the case that a user can design an application with any desired set of patterns, but will be responsible for implementing those patterns that are not directly supported by the tool.



(a) An example of a general mesh with three partitions.

(b) A regular, fully–toroidal mesh with four partitions.

Figure 3.2: Examples of the Mesh design pattern.

3.2.2 From Design Patterns to Design Pattern Templates

As noted earlier, a design pattern does not represent a single solution to a given design problem, but rather embodies a family of potential solutions. The main structure of the solution remains the same, but several variations exist to tailor the pattern to the specific design problem at hand.

To incorporate this idea of a pattern being a family of solutions, the PDP process includes the concept of a *design pattern template*. The template is a design construct that represents the basic structure of the pattern, but includes pattern template parameters to further specialize the template. After the user has determined the most appropriate parallel design pattern for a problem, the corresponding pattern template is selected. The values of the pattern template parameters refine the template and select the member of the family of patterns that is best suited for the problem. It is important to note, though, that a template may be restricted to a subset of the complete family. The parameters may not allow certain members to be selected.

For the reaction–diffusion example, a user would select the Mesh pattern template using the CO_2P_3S user interface, shown in Figure 3.3. The Mesh design pattern template in CO_2P_3S supports mesh computations with regular, rectangular two–dimensional data. It does not support general mesh data (such as that shown in Figure 3.2(a)). The parameters for this template, specified through the interface, are:

1. The name for the class representing the mesh. In this application, the name is `RDMesh`. The programmer uses this class for two purposes. First, this class is instantiated to create all of the framework objects that implement the pattern. Second, instances of this class are used to launch the computation.

2. The class name for the elements that populate the two–dimensional mesh structure. For this problem, the element class is called `MorphogenPair`. The generated framework code for the Mesh template will create a two–dimensional structure holding instances of these objects. The mesh computation is executed on these objects, so this class holds application–specific state for each element and provides a set of hook methods for implementing the computation with respect to an individual element (or node in the graph of Figure 3.2(b)). The hook methods are described in Section 3.2.3. The `MorphogenPair` class in this example holds two morphogens for a region of the simulated surface, and defines methods for updating the concentration values of these morphogens using the simulation parameters.

   In addition, the user can also specify the superclass of this element class, to fit the class into an existing inheritance hierarchy. In this application, the mesh element class has no superclass so an appropriate default is used. Since CO_2P_3S generates Java code, the default is `Object`.

3. The topology of the mesh. This parameter determines how elements on the edge of the mesh structure are treated. A fully–toroidal mesh is used, as shown in Figure 3.4(a), so that the generated textures can be tiled. The Mesh template also supports non–toroidal, horizontal–toroidal, and vertical–toroidal topologies.

4. The maximum number of neighbouring elements that each mesh element uses to calculate its new value. This reaction–diffusion simulation diffuses the morphogens in the horizontal and vertical directions. Thus, the user picks a four–point mesh from the dialog in Figure 3.4(b). Eight–point meshes, which include neighbouring elements on the diagonals, are also supported for problems like the Game of Life or more complex reaction–diffusion simulations.

5. The amount of synchronization for the computation. A mesh computation can be ordered or chaotic. In an ordered mesh, each element of the mesh waits for the remaining elements to compute their new value before starting the next iteration. In a chaotic mesh, each element starts its next iteration as soon as it can, possibly using old data from neighbours that have not completed earlier iterations. The specification of the reaction–diffusion simulation requires Jacobi iteration to generate the correct results, which requires that the mesh be ordered. If a chaotic mesh is selected then this corresponds to using Gauss–Seidel iteration.

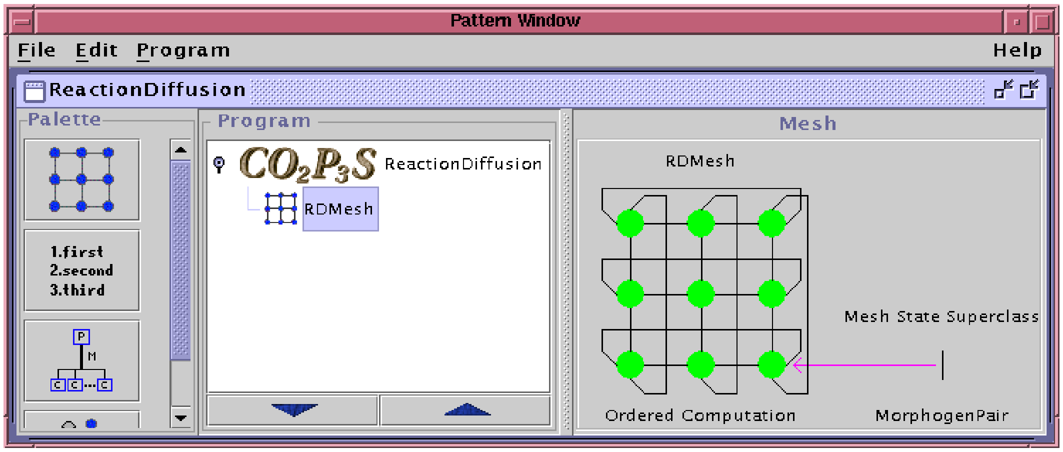
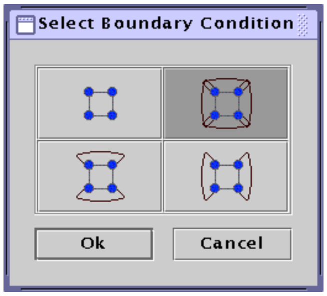
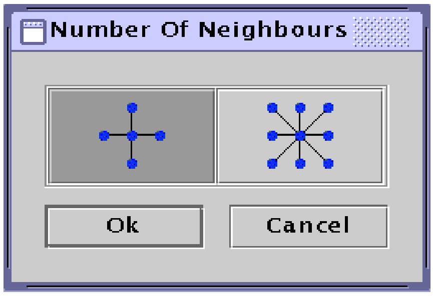


Figure 3.3: A screenshot of CO_2P_3S showing the reaction–diffusion application using the Mesh pattern template.



(a) Mesh topologies.



(b) The number of neighbours.

Figure 3.4: Examples of parameters for the Mesh pattern template.

This may produce a texture that looks "good enough" for its intended use and takes less time to compute, although this would be changing the original semantics of the problem. We have not investigated a chaotic solution to this problem.

3.2.3 From Pattern Templates to Frameworks - The Patterns Layer

After the programmer has specified values for the design pattern template parameters, the system gives the template to a code generator to create a framework for the pattern, which is customized by the pattern template parameter values. This framework is another example of an object–oriented framework, with a set of abstract classes defining the application–independent flow of control and a set of concrete subclasses that define the application–specific functionality. This application–specific functionality is inserted into the flow of control by overriding hook methods invoked from the abstract classes. The generated framework includes both abstract classes and concrete subclasses. The concrete classes available to the user include stubs with default implementations of the hook methods. This is important as the values of the pattern template parameters can affect the set of available hook methods and their signatures. Generating stubs for these methods saves the user from having to derive this information from the parameter values.

The frameworks generated at the Patterns Layer exhibit certain characteristics to make it easier to create a structurally correct parallel program. First, to ensure correctness, the structural code is hidden from the user so that it cannot be modified. Thus, the user starts with a correct parallel structure for the selected pattern, and cannot introduce structural errors at this layer. Second, as well as defining the flow of control, the abstract structural classes contain all necessary parallel code, including communication and synchronization. Specifically, the structure is implemented such that the hook methods do not require any parallel code. The hook methods can be implemented as normal, sequential methods. Last, the default hook methods implementations allow the frameworks to be compiled and run immediately after they are generated. This allows an application to be developed incrementally.

An additional benefit to hiding the structural code is the simplification of the frameworks from the user's perspective. Only those abstractions that are germane to implementing a problem need to be exposed. The user will not be distracted by any abstractions or infrastructure for supporting the parallelism in the selected pattern templates.

Another part of any tool implementing the PDP process is to track the last layer that the user was working at for each template in a program, and display that progress to the user in some way. In $\mathrm{CO_2P_3S}$, this is accomplished using colours in the graphical representation of the pattern templates. The green in the nodes in the graphical representation of the Mesh template in Figure 3.3 indicates that this template has been used to generate framework code at the Patterns Layer. If the framework code has not been generated or is out of date (because the user has changed the value of a parameter), the nodes are red. This ability to track which layer the user last used when working with a template is vital for the lower layers, as any changes made there may not be reflected by any changes at the code normally visible at higher layers. For example, a change in the parallel structure may not cause any changes to any of the classes that the user can access at this layer. If the tool does not provide this information, users may forget about this structural change and get unexpected results.

The generated framework code, together with the hook method code, comprise the *Patterns Layer Code*. The Patterns Layer is complete. The user now has a complete structurally correct parallel program. This program can be compiled and executed.

It is important to again emphasize the difference between a structurally correct parallel program and a semantically correct parallel program. At this stage of development, the user has framework code that correctly implements the structure indicated by the combination of the selected design pattern templates and their parameters. The program is considered to be structurally correct. However, there is no guarantee that this framework code can be used as the basis for a semantically correct parallel program. The selected template may not provide the parallelism needed for the problem or one of the parameters of the template may have the wrong value. Even if the templates and their parameters are correct, there is no guarantee that after the user edits the hook methods, the code correctly implements the problem. There may be errors in the user code, or the code may have been inserted in an inappropriate hook method (and is thus being executed by the framework at the wrong time).

Consider the reaction–diffusion example in CO_2P_3S. After specifying the parameters for the Mesh pattern template, CO_2P_3S generates framework code for the template. The structural part of this code is hidden from the user so that it cannot be compromised. Instead, the user's view of the Mesh framework consists only of the mesh element class. This class has the following responsibilities:

- Creating an instance of itself by applying a user–supplied initializer object.
- Determining if it has reached its final value.
- Computing its new value based on its current value and the values of the available neighbouring elements.
- Gathering its final value into its final form by applying a user–supplied reducer object.

The rest of the framework code uses these primitive operations to create a complete mesh computation. Note that the user's responsibilities are oriented towards solving the problem, not supporting the parallelism in the generated framework. The user can concentrate on how to implement these responsibilities for each mesh element and not on how the framework will parallelize the computation.

To make this example more concrete, consider the main loop of the mesh computation for each process, given in Figure 3.5. This code is part of the framework structure and is not available to the user at the Pattern Layer, but it demonstrates the flow of control through the application. In particular, it shows the order in which (most of the) hook methods are invoked. Some parts of the computation are not present in this figure, notably data initialization (which is done before the code in `meshMethod()`) and result gathering (which is done after). The `initialize()`, `prepare()`, and `postProcess()` methods invoke a method with the same name to each element in the partition. The responsibilities of each of these methods is given in Table 3.1. The `notDone()` method invokes a method with the same name on each element in the local

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

Figure 3.5: The main execution loop for each partition in the Mesh framework. The italicized methods have corresponding hook methods that can be implemented by the user.

partition, but also combines the return values for all processes to indicate if the computation has finished computing. The `barrier()` call implements the needed synchronization between the preprocessing of the mesh elements (in the `prepare()` method) from the computation of new values (in the `operate()` method, which is discussed later). In a chaotic mesh, this call is not present. In a distributed memory environment, code to explicitly exchange boundaries would appear before this barrier call. Since the frameworks generated by CO_2P_3S use threads and shared memory, this exchange is implicit.

The `operate()` method in Figure 3.5 is different in that it does not invoke an identically named method on each of the mesh elements in its partition. Rather, this method invokes one of nine possible operation methods, listed in Figure 3.6. The invoked method for a given mesh element is determined by the location of the mesh element in the mesh data and the topology of the mesh, as shown in Figure 3.7. The complete set of neighbours for a given mesh element is determined by the number of neighbours used in the mesh computation. The methods in Figure 3.6 are for a four–point mesh; an eight–point mesh includes neighbours along the diagonal as well, changing the signatures of the operation methods. The Mesh framework enumerates over the mesh elements in a partition, determines the correct operation method, and invokes that method with the correct neighbour information. To make the framework easier to use, stubs are generated in the mesh element class for only those operation methods that may be invoked. This saves the user from having to determine both the correct subset of operation methods that must be implemented and the proper signatures for those methods.

Using the supplied hook methods for the Mesh framework, the reaction–diffusion is implemented as follows. The initializer object for the constructor of the `MorphogenPair` class is a random number generator so that the two instances of the morphogens can be created with random initial concentrations. Note that the two–dimensional mesh data is created sequentially to guaran-

Table 3.1: The hook methods (except the operation methods, listed in Figure 3.6) for the instance of the Mesh framework used for the reaction–diffusion example.

Hook methods with signature	Implemented responsibility

`MorphogenPair(int i,`
`int j,`
`int surfaceWidth,`
`int surfaceHeight,`
`Object initializer) ;`
	This method constructs a single instance of the mesh element class at the location (i, j) of the two–dimensional structure by applying the given initializer object supplied by the programmer. The mesh elements are created in another part of the generated structural code that executes before the threads are created.

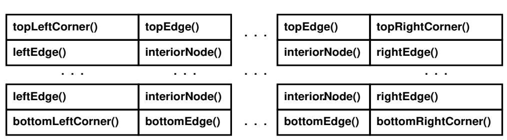
`void initialize() ;`
`void prepare() ;`
`void postProcess() ;`
	These methods allow application code to be inserted at various points in the mesh computation. The `barrier()` call in Figure 3.5 implements the necessary synchronization in the mesh structure, and is not the user's responsibility. It ensures that all of the threads have finished any preprocessing for an iteration (in the `prepare()` call) before they compute the new value for the mesh elements.

`boolean notDone() ;`	This method evaluates the termination condition for a single mesh element. The computation continues until all mesh elements return false. The return value of the `notDone()` method in Figure 3.5 is the combined result for each of the calls of this method on the individual mesh elements.

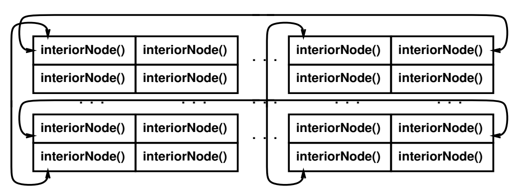
`void reduce(int i,`
`int j,`
`int surfaceWidth,`
`int surfaceHeight,`
`Object reducer) ;`
	This method is responsible for applying the user–supplied reducer object to gather the results of the mesh computation after it has completed. Like the constructor, the calls to this method are in another part of the structural code that executes after all of the threads have finished computing.

```
void topLeftCorner(MorphogenPair right,
        MorphogenPair down) ;
void topEdge(MorphogenPair left, MorphogenPair right,
        MorphogenPair down) ;
void topRightCorner(MorphogenPair left,
        MorphogenPair down) ;
void leftEdge(MorphogenPair right, MorphogenPair up,
        MorphogenPair down) ;
void interiorNode(MorphogenPair right, MorphogenPair left,
        MorphogenPair up, MorphogenPair down) ;
void rightEdge(MorphogenPair left, MorphogenPair up,
        MorphogenPair down) ;
void bottomLeftCorner(MorphogenPair right,
        MorphogenPair up) ;
void bottomEdge(MorphogenPair left, MorphogenPair right,
        MorphogenPair up) ;
void bottomRightCorner(MorphogenPair left,
        MorphogenPair up) ;
```

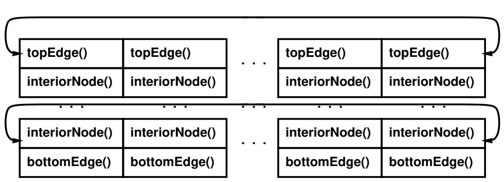
Figure 3.6: The operation methods for the mesh computation in the four–point Mesh framework for the reaction–diffusion example.



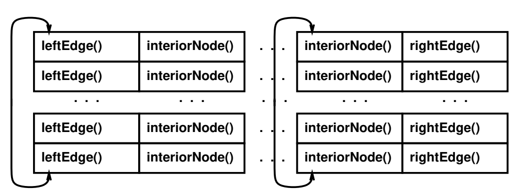
(a) Non–toroidal operations.



(b) Fully–toroidal operations.



(c) Horizontal–toroidal operations.



(d) Vertical–toroidal operations.

Figure 3.7: The operation method calls for mesh elements in different positions, for each of the topologies from Figure 3.4(a). Note that the above diagrams represent the complete mesh data, not a single partition.

tee that the initialization is reproducible. It is important to note that each morphogen keeps two concentration values: a read value that holds the value from the previous iteration, and a write value for the result of the current iteration. The default `initialize()` method stub, which does nothing, is used as there is no additional initialization needed for each mesh element before the computation starts. The `notDone()` method for each element checks for convergence by checking the difference between the read and write concentration values. The simulation ends when the changes in all morphogen concentrations have fallen below a given threshold. The `prepare()` method updates the read concentration value for the two morphogens with the write value. This is required because the read value is used by neighbouring mesh elements when calculating new values. To ensure that this update is completed before new values are computed, the mesh needs to have the ordered parameter selected, which inserts the call to `barrier()` in the main loop. Without this synchronization, it would be possible for a mesh element to compute its new value based on outdated neighbour data, which yields incorrect results for this algorithm. Given that this mesh is fully toroidal, the only operation method that need be implemented is `interiorNode()`. In fact, since the user selected the fully–toroidal parameter for the topology of the mesh, it is the only operation method stub that is generated. This method computes a new concentration value for each morphogen in the mesh element based on all four neighbouring elements. The topology ensures that these four neighbours exist for all mesh elements. The `postProcess()` method performs one final update of the read values from write values before the results are gathered. Finally, the `reduce()` method, not shown in Figure 3.5, gathers the final result into an output array that can be used to display the final texture. This method is invoked sequentially on the entire mesh data structure after the computation has finished. All of the user code for implementing this program is given in Appendix A.

In addition to the hook method implementations and required collaborator classes (such as the `Morphogen` class for a single morphogen, which is taken from the sequential version of the problem), the user must create a mainline method to create and launch the computation. To help, the CO_2P_3S system generates a sample mainline method that the user can modify for the specific computation, given in Figure 3.8(a). The simplified mainline for the reaction–diffusion example is given in Figure 3.8(b). This code is not the default mainline generated by CO_2P_3S, but rather is a version modified for this program. Recall that the name of the class representing the mesh is `RDMesh`. The mainline takes the size of the mesh data from the first two command line arguments and the number of horizontal and vertical partitions from the third and fourth arguments. The data is evenly distributed over the partitions and each partition is assigned to a different process. The initializer and reducer arguments needed for the sample problem, a random number generator and a two–dimensional array of final concentration values, are created next. The mesh framework is instantiated by creating an object of type `RDMesh` with the needed constructor parameters. The computation is started by invoking the `launch()` method on the mesh object. When this method returns, the computation will be finished and the final results will be available in the reducer object.

If a parallel application consists of several patterns composed together, the user will also need to write code to take the output from one pattern and pass it to the input of the next. Part of this process may be handled by additional patterns available within the tool. In addition, the tool should have a method for developing new classes or incorporating existing classes that can be used in a parallel application. The latter, combined with the property that hook methods contain sequential code, can promote the reuse of existing sequential code. For instance, in the reaction–diffusion problem, the `MorphogenPair` class includes two instances of the `Morphogen` class, which is reused directly from the sequential code.

Once the hook methods and other necessary code are implemented, the programmer has a parallel program that can be executed on a parallel machine. While the user can introduce logic errors in the application–dependent code, the parallel structure cannot be compromised. Thus, the user can concentrate on the logic in the application and not on the parallel framework code that executes it. This is an improvement over other parallel programming systems that force users to consider parallel and concurrency issues during development by requiring them to implement the parallel structure of their applications.

The two key aspects of the Patterns Layer are separation and correctness. Separation refers to the clear distinction between the application–independent parallel structural code and the application–specific sequential hook method code. Maintaining this separation allows the structural code to evolve independently of the application code. Correctness refers to the ability of the user to write correct parallel programs. The remainder of this section examines these two aspects in more detail, based on the reaction–diffusion example.

Separation

Separation is achieved by generating framework code for the supported patterns. Application frameworks are built with the structural code, defining the flow of control through a program, implemented in abstract classes. An application is created by subclassing these abstract classes to implement methods that are invoked by the structural code. This can also be considered an application of the Generation Gap pattern [108], which separates generated code from application code by placing the generated code into abstract classes. This separation allows the application–independent classes to be regenerated without affecting the application–specific classes. Generating framework code from pattern specifications solves the same problem. However, the code generated at the Patterns Layer is intended to be modified at lower layers of abstraction, which is not part of the intent of the Generation Gap pattern.

For instance, consider the topology parameter in the Mesh pattern tem-

```
public static void main(String[] argv) {
    int dataWidth = 1 ;
    int dataHeight = 1 ;
    int meshWidth = 100 ;
    int meshHeight = 100 ;
    Object initializer = null ;
    Object reducer = null ;
    RDMesh mesh = new RDMesh(surfaceWidth, surfaceHeight,
            meshWidth, meshHeight, initializer, reducer) ;
    mesh.launch() ;
    // When mesh.launch() returns, the results will be
    // available via the reducer object.
} /* main */
```

(a) The mainline code generated for the framework for the Mesh pattern template.

```
public static void main(String[] argv) {
    int dataWidth = Integer.parseInt(argv[0]) ;
    int dataHeight = Integer.parseInt(argv[1]) ;
    int meshWidth = Integer.parseInt(argv[2]) ;
    int meshHeight = Integer.parseInt(argv[3]) ;
    Random initializer = new Random() ;
    double[][] reducer =
            new double[dataWidth][dataHeight] ;
    RDMesh mesh = new RDMesh(surfaceWidth, surfaceHeight,
            meshWidth, meshHeight, initializer, reducer) ;
    mesh.launch() ;
    // When mesh.launch() returns, the results will be
    // available via the reducer object.
} /* main */
```

(b) Simplified mainline code for the reaction–diffusion example.

Figure 3.8: Generated and modified mainline code for the framework generated for the Mesh pattern template.

plate. This parameter determines the set of operation methods that may be applied to a mesh element. If this parameter is changed, it should be possible to regenerate the framework without losing the implementation of any operation method that can still be invoked. The programmer should only have to implement any new operation methods that can be invoked with the new topology. Similarly, it should be possible for the number of neighbouring elements to change without losing application code. The programmer will need to modify the operation methods to deal with the different set of neighbours, but the remaining code should not be affected. This allows the structural code to evolve and change relatively independently of the application code.

In general, changing parameter values should yield few code changes. While the structural code changes, the interface between it and the application–specific classes should be relatively stable. Changing patterns, on the other hand, could give rise to substantial changes, as the new interface may be considerably different. The interface between the structural code and application–specific code depends on the problem that the framework is trying to solve, or rather on the intent of the pattern that the framework is implementing in our case. Patterns with different intents can have substantially different interfaces to application–specific code.

Correctness

The Patterns Layer addresses correctness in several ways. These can be broken down into three categories: parallel structural correctness, encapsulation of structural code, and framework usability.

Parallel structural correctness is achieved by generating correct parallel structural code in the abstract framework classes. Most parallel programming systems provide a high–level programming model, but still require the programmer to create the complete structure of an application. Although the programming model abstracts out some of the difficulties in writing parallel programs, implementing the structure is an error–prone process. For instance, in a message–passing environment, the sending and receiving of messages between different processes must match or processes may incorrectly block forever. In a shared–memory environment, incorrect synchronization may result in sporadic, non–deterministic errors. This problem is exacerbated by the lack of good debugging tools and techniques. Given that writing this structure is often difficult, it seems clear that tool support should be directed here. This is the premise behind the Patterns Layer. Given a parallel design pattern template, it is possible to generate a complete, correct parallel structure. The user does not have to write or debug this code.

Another aspect to correctness at the Patterns Layer is the encapsulation of the structural code such that it cannot be modified. At this layer, the user can only supply sequential hook method implementations. This encapsulation prevents the user from accidently introducing structural errors into the parallel code. Further, the frameworks should not rely on the user to correctly implement any parallel code in the hook methods. This parallel code is part of the structure of a parallel pattern, and should not be of concern to the user at the Patterns Layer. As a result, the frameworks do not rely on hook methods to properly implement any parallel code in order to work correctly. The structural code includes all necessary parallel code to ensure that the hook methods can be written as normal, sequential methods.

For instance, consider the barrier synchronization required for the ordered mesh (in Figure 3.5). The user specifies this synchronization via a Mesh pattern template parameter, which affects the generated structural framework code. The user is not responsible for inserting or implementing this code; it is encapsulated in the framework. Further, the user cannot accidently remove this barrier.

The final correctness aspect, framework usability, refers to reducing the probability of user errors when implementing an application using the generated frameworks. Clearly, parallel structural correctness is part of this concern. However, learning to write an application with a framework also entails a learning curve. A programmer must understand three issues about a framework in order to use it: how the hook methods in the framework can be used to specialize the behaviour of the structure, which hook methods need to be implemented for a given application (and, conversely, which can be left with their default implementations), and how the different classes in the framework must be composed into a complete application. Adequate documentation is the solution to the first issue and part of the second. The remainder of the second issue and the third can be addressed through conditional code generation, again based on the parameters for the design pattern templates.

In some cases, a framework may present hook methods that may not be used by the structural code. This can happen when the programmer selects a different strategy than the framework uses in some part of its computation. For example, a different framework for the mesh may include all nine operation methods as hook methods, but only invoke a subset of them based on the mesh topology. The programmer would then be responsible for determining the correct subset of operation methods for the application. This can be the source of errors if the programmer implements the incorrect methods. Also, if the topology changes then it is possible for the programmer to forget to implement any additional methods that may now be invoked. To prevent these errors, CO_2P_3S generates concrete classes that include stubs for only the hook methods that can be invoked. The topology for reaction–diffusion example is fully–toroidal, so the only operation method that can be applied to a mesh element is `interiorNode()`. When the framework is generated, the mesh element class only includes a stub for this method and none of the others. The user does not see any irrelevant hook methods, and so does not have to derive the set of methods that need to be implemented based on the template parameters. Instead, if the stub appears in the mesh element class, the method can be invoked and should be implemented.[3] Further, breaking down the mesh operation into nine separate methods saves the user from having to write a single, large operation method that must take the different topologies into account. Although a single operation method would cut down on the number of hook methods in the framework, the implementation of this method would unnecessarily duplicate the topology information. If the topology changes, it may not be clear if the single operation method is out of date. With multiple operation methods, stubs for new methods appear in the mesh element class and make it clear what new conditions are now possible and must be accounted for. This is an example of using code generation to reduce the probability of user errors.

Another problem when building applications from frameworks is the correct composition of the framework classes. The user not only needs to correctly subclass the structural classes to implement hook methods, but also needs to correctly compose both these subclasses and other collaborating classes into a complete application. These other collaborating classes may implement policies or options in the framework. Again, the code generator can generate code that automatically handles this composition without user intervention, based on pattern template parameters. For instance, in the Mesh framework, an instance of the Strategy pattern [37] is used to invoke the correct operation methods on the mesh elements. In a normal framework, the user would have to implement the mesh element class and then compose the other framework classes into an application. This composition would include creating and using the specific strategy for the topology. Since the topology is a Mesh pattern template parameter, the appropriate Strategy object can be created and automatically composed into the framework. Further, the user interface can be responsible for ensuring that a valid topology is selected.

The composition of framework classes can be further simplified by restricting the user's view of the framework. The user is aware of only those classes that are germane to creating an application. The remaining classes and responsibilities are encapsulated by the pattern template, including their composition into a complete application. This clarifies the responsibilities of the user by removing unnecessary distractions from the development process. This is analogous to how behavioural patterns encapsulate complex object interactions. These interactions are implicit when applying a behavioural pattern, allowing the user to abstract out this detail and concentrate on the interaction between the objects [37].

[3] It is, of course, not necessary to implement all hook methods, either to develop a program incrementally or sometimes to achieve desired results. For instance, to implement a mesh computation where the elements on the edge of the data structure are constant and should not be updated, one can use an instance of the Mesh framework with a non–toroidal topology and only implement the `interiorNode()` operation method.

3.2.4 From Frameworks to Parallel Programs I - The Intermediate Code Layer

Ideally, the Patterns Layer would be sufficient to create an efficient parallel program. Unfortunately, this is not possible for two reasons. First, it would require the set of available pattern templates to cover the complete spectrum of parallel programming structures, which is not possible. Second, not all aspects of the pattern templates can be specialized. The parameters for a given template may not allow certain alternative pattern implementations to be specified. For instance, the Mesh pattern template does not support general mesh computations. Further, the resulting frameworks for a given template are conservatively correct, as the framework must be applicable to a broad range of applications using the selected structure. Consequently, the generated frameworks may include synchronization that may not be necessary in a particular application, or may serialize certain parts of a computation that can be executed in parallel. As well, the framework may include generic code that can be simplified for a given application.

To address these limitations, the PDP process includes additional layers to support lower–level parallel programming and performance tuning. These layers are also a response to the closed nature of existing approaches in parallel programming systems. With the exception of DPnDP, parallel systems provide a single, closed programming model for writing programs. These models work well for certain problem domains, but any program outside of that domain cannot be implemented efficiently. There is no provision for circumventing the model when a problem requires it. Even for problems within the domain, though, the performance of a program is limited by the overhead introduced by the programming model. Unfortunately, the closed nature of the programming model in these systems prevents the programmer from removing this overhead and improving the program. If the performance of an application is not acceptable, the programmer usually has no choice but to find another system that yields a more efficient program.

The second layer in the PDP process is the Intermediate Code Layer. This layer provides a high–level, explicitly parallel object–oriented programming language. This language may be an extension of an existing language such as Java or C++ that includes high–level parallel constructs such as barriers and parallel loops. The implementation of these new constructs, though, is not accessible. In addition, this layer opens up the structural framework code, making it available to the user. This code, written in our intermediate language, can be modified or rewritten by the user.

The Intermediate Code Layer is intended to be a gentle transition from the high–level patterns and frameworks at the Patterns Layer to the low–level run–time support code. The structural framework code is supplied at a high level of abstraction, making it easier to modify. These modifications can be used to make application–specific changes or optimizations to the structure. The high level of abstraction allows novice programmers to consider these changes.

For example, at the Patterns Layer the user can only see two classes in the Mesh framework: the mesh class, which is only instantiated and used, and the mesh element class, which implements a set of hook methods with respect to the individual data element. A complete parallel mesh computation entails the following responsibilities:

1. Creating the two–dimensional data structure and populating it with mesh elements.

2. Creating the different partitions from the mesh data, assigning partitions to processes, and starting the processes.

3. The mesh computation itself. This is a loop consisting of the following steps, executed by each process for its local partition:

   (a) Determine if the computation has finished (*i.e.* if the computed values have converged in each partition).

   (b) Exchange the boundaries of the partition with neighbouring partitions.

   (c) Compute new values for each node in the partition.

4. Gathering final results.

At the Patterns Layer, the most important aspect of the Mesh framework is that all of the responsibilities above are implied by the Mesh pattern template, and as such can be handled automatically by the system. The four responsibilities above define the complete mesh computation in a generic way. The system uses the template parameters, hook methods in the mesh element class, and the constructor parameters for the framework to transform this generic structure into the specific mesh computation desired by the user.

At the Intermediate Code Layer, the implementation of the responsibilities is opened. For the reaction–diffusion example, the complete implementation of the generated framework is given in Figure 3.9. The classes that are visible to the user at the Patterns Layer are shaded in gray. The `MorphogenPair` class is the mesh element class, with user–supplied hook method implementations. The `RDMesh` class is the class that is instantiated to create and launch the computation. The `RDMesh` class creates the two–dimensional data (the `BoundedMorphogenPairArray` object, which consists of `MorphogenPair` objects), the partitions of the mesh data (the `RDMeshBlock` objects), and the threads for the computation. Since CO_2P_3S frameworks are Java code with shared memory, the partitions are actually implemented in the bounded array objects by sharing one copy of the large data set and providing accessor methods that use offsets to provide access to a subset of the data. Finally, the `RDMeshBlockStrategy` class is part of the Strategy pattern [37] used during the computation of new values to determine which operation method to invoke and which neighbours to pass as parameters.

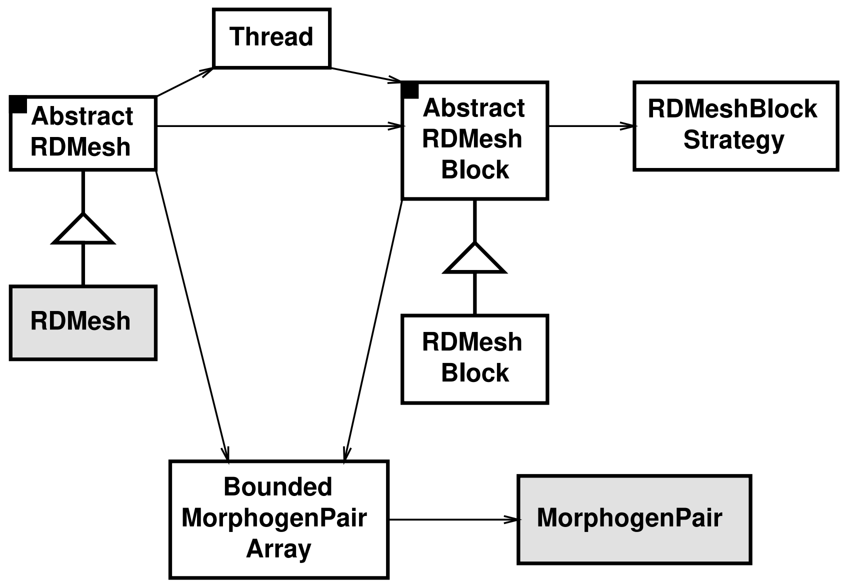


Figure 3.9: The complete implementation of the Mesh framework. The classes that are visible to the user at the Patterns Layer are shaded in gray.

As a concrete example of a task for the Intermediate Code Layer, consider the gathering of the end result in the Mesh framework in CO_2P_3S. This gathering is done by applying a reducer object to each mesh element sequentially, after the final values have been computed. This is necessary because, in general, it is not possible to guarantee that this gathering can be done in parallel. In the reaction–diffusion example, the reduction copies the final concentration value into an output array. However, in this case, the operation can be done in parallel as each value in copied to a unique array element. Since the programmer now has access to the structural code in the Mesh framework, this change can be made.

To make the process of changing the structure of the generated framework easier still, this structural code can itself be implemented as a framework, complete with hook methods. This makes it easier to make simple changes to the structural code. If the changes are beyond the scope of the hook methods, the code in the abstract classes can be edited directly. The drawback to this approach is that the code is overwritten if the framework is regenerated.

An important feature of the Intermediate Code Layer is that the intermediate language is a parallel programming language in its own right and can be used to write parallel applications. In fact, a user can decide not to use any patterns at the Patterns Layer and instead write a complete application at the Intermediate Code Layer. This is particularly useful if no pattern in the tool provides the parallel structure needed for an application. For example, a programmer could use the intermediate language to solve a general mesh computation, which is not supported in CO_2P_3S.

It must be noted that any modifications to the structure can result in an incorrect program. Starting with this layer, the programmer is responsible for the correctness of both the structural code and the application code. However, if errors are introduced into the structure, it is possible to explicitly regenerate the structure and begin again with the Pattern Layer Code.

3.2.5 From Frameworks to Parallel Programs II - The Native Code Layer

If the performance of the application is not acceptable after modifying its structure at the Intermediate Code Layer, it is possible to further improve performance by modifying the implementation of the primitives using the Native Code Layer. Where the Intermediate Code Layer permits application–specific tuning, the Native Code Layer permits system–specific tuning. This is the last abstraction in the PDP process.

This layer presents the underlying object–oriented programming language together with any libraries used to provide the abstractions at the higher levels. These libraries also include any architecture–dependent attributes of the system. All of this support code can be modified by the programmer. For example, barrier synchronization has several implementations that take into account parallel architecture (shared memory versus distributed memory) and the amount of expected contention. The implementation of the barrier can be rewritten with another version that is more appropriate for the application and its execution environment.

3.3 Benefits of the PDP Process

The first and most obvious benefit to the PDP process is the separation of concerns inherent in layered software. The layers in this process correspond the different phases of program development, from initial development (the Patterns Layer) to high–level program tuning and structural changes (the Intermediate Code Layer) to low–level system tuning (the Native Code Layer). Each development phase can benefit from different abstractions, which can now be applied separately.

The PDP process offers a unique approach to building correct parallel applications through code generation and encapsulation at the Patterns Layer. Current parallel programming systems generate support code for an application, but still require the user to implement the structure of an application. However, writing and debugging this structure is often the most difficult part of parallel programming. By generating a debugged version of this structural code, the Patterns Layer promotes the rapid development of parallel programs. The user does not need to write and debug the structural code, and can instead concentrate on application logic. Further, encapsulating the structural code into a set of abstract classes that are not available for modification prevents users from accidently introducing errors. This encapsulation leads to another useful property of the Patterns Layer: the decomposition of a parallel application into sequential and parallel code. The abstract structure contains all the needed parallel and synchronization code such that application–specific code is written as sequential code. This allows for the reuse of existing sequential code to the extent that is possible in an application.

In the Patterns Layer of the PDP process, framework code is specifically generated for the pattern template, rather than reusing a single implementation. In our case, generating code has several advantages that would be difficult to achieve otherwise. These advantages are:

- With pattern template parameters, it is possible to add application–specific methods to the generated framework code. This is not necessary in the Mesh template, as its interfaces are fixed. However, the frameworks for other pattern templates, the Distributor (Section 4.2.1) and the Phases (Section 4.2.2), include application–specific methods in their interfaces. With a single, static pattern template framework, these methods would be difficult to incorporate into a program.

- Generating the code simplifies the class and object structure of the resulting framework. The framework implements the pattern structure indicated by the pattern template parameter values supplied by the user rather than being a generic implementation. A single implementation would need to introduce indirection into the code to accommodate different pattern variations. For instance, the Strategy pattern [37] could be used to specify policies in the framework. At the Patterns Layer, these policies are good examples of pattern template parameters, which guide the framework generation process. Rather than including a Strategy object, the correct code can be generated and inserted directly into the framework structure, reducing the number of objects and classes in the code.

- The generated frameworks may have better performance. By reducing the amount of indirection in the framework (as indicated above), it may execute faster.

- The structural code is further simplified by only including relevant hook methods in the framework. For example, in the Mesh framework, we only include the subset of operations methods that are relevant. Otherwise, it would be necessary to include all possible operation methods in the structural code (18 in total), which might be confusing to the user. In addition, the signatures for the relevant hook methods are included in the concrete subclasses that are supplied to the programmer. This saves the user from having to enter these signatures and reduces the possibility of introducing compilation or logic errors into the hook method code.

- Additional support code can be generated and tailored to the pattern structure. For example, the code for instantiating the framework (since the structure is encapsulated at the Patterns Layer) is generated specifically for the selected pattern structure.

- Generating code for each pattern template allows each framework to be treated in isolation at lower layers. Each can be optimized independently. Otherwise, it would be difficult to tune programs as any structural changes would have to be applicable to all uses of the framework in the application.

Further, the combination of the layers in the PDP process promote openness to allow performance tuning and structural modifications, which is currently lacking in most research systems. Unlike DPnDP, which provides limited access to the low–level facilities, the PDP process provides several layers of abstraction to gradually expose implementation details. Dividing performance tuning over multiple abstractions offers usability advantages. A programmer can make changes at an appropriate layer of abstraction based on what is being tuned. For instance, simple structural changes should not involve details on how synchronization structures are implemented, as this detail is not relevant to the task. Further, users can select an abstraction based on how comfortable they are with it. Novice parallel programmers may wish to work at the Intermediate Code Layer, while experienced programmers may want access to the complete system while tuning. By providing both abstractions, the PDP process provides performance tuning opportunities for all users. This graduated approach leads to a system where the performance of a parallel program is commensurate with programmer effort. Tools implementing this process will provide a flexible environment for building parallel object–oriented programs.

3.4 System Developer Issues in the PDP Process

To this point in this chapter, the focus has been on how a programmer uses a tool based on the PDP process to implement a parallel program. However, this process is also intended to guide parallel systems developers in creating new tools. This section examines several issues that system developers must address in any tool that supports the PDP process.

3.4.1 Creating the Design Pattern Templates

The Patterns Layer provides a set of design pattern templates that are used by the programmer to describe the parallel structure of a program. However, these templates must first be created by the tool developer. Many pattern templates can be created for use in CO_2P_3S, and each template brings with it a number of questions that need to be answered. Is the template necessary, or is it already addressed by existing templates? How useful will the template be in practice? How general should it be? What parameters should be available for the template? What restrictions should it have? How efficiently can it be implemented? These issues are for the tool developer, not the tool user. This section examines the process of creating design pattern templates.

Recall that a pattern is a description of a good solution to a recurring design problem. These solutions are the direct result of experience in design by experts in the field. The primary purpose of a pattern is to describe and discuss the benefits and drawbacks of the proposed solution and to provide a common lexicon of design terms for software developers. From this abstract idea, the more concrete design pattern template is created.

Design pattern templates take advantage of the commonality of different design pattern implementations. Although part of a pattern description focuses on alternative implementations, each implementation shares a set of common features. For example, the Composite pattern [37] lists nine implementation concerns that can shape the final use of the pattern. However, the basic strategy of maintaining a tree-like container structure based on compositions of containers and leaf nodes is common across all implementations.

The first step in creating a pattern template is to identify this common structure in the pattern. For instance, in the Mesh pattern, the common structure is an iterative computation over a set of elements distributed over different processes, where each element uses the values of some neighbourhood around it to determine its new value.

Once the common structure has been identified, the pattern template can be refined to narrow the scope of the pattern. For instance, the Mesh pattern applies to general, undirected graphs. However, the Mesh pattern template in CO_2P_3S narrows the scope of this pattern to regular, rectangular two–dimensional meshes (from Figure 3.2(b)) where each element computes its new value using data from its immediate neighbours. Imposing this structure on the mesh data makes it easier to represent the data (using a two–dimensional array) and generate partitions (by blocking the data). It further simplifies the computation as the neighbourhood around a mesh element is fixed. The next section shows how we can take advantage of this structure when generating code for the Mesh design pattern template.

Finally, after the fixed portion of the template has been determined, the parameters to the template must be decided upon. There are two considerations that must be addressed in selecting the template parameters. First, they should permit a variety of implementation options with respect to the fixed portions of the template. For instance, given a Mesh pattern template that supports regular, rectangular two–dimensional data, the pattern parameters and supported hook methods can be used to create a number of different mesh computations. In contrast, a general mesh template would need additional parameters or hook methods for partitioning the data across processes.

The second consideration is related to the amount of effort needed to change parameters for a template, particularly after framework code has been generated. The framework must be regenerated if any of the template parameter values are changed, since the parameters dictate the structure of the framework. For some parameters, this effort is warranted and even necessary. For example, changing the boundary conditions in the Mesh template will add and remove some of the operation methods from the mesh element class, since the boundary conditions are a fundamental part of the structure of a mesh computation. In addition, the structural code must be modified to recognize the new conditions and call the proper operation methods. For other parameters, though, this regeneration unnecessarily limits the application built using the framework. The size of the mesh data and the number of partitions in the Mesh template are examples of parameters that should not require regeneration. Fixing these values in the structural part of the framework makes it too difficult to experiment with different numbers of processors and different data sets. In addition, it is straight–forward to generate framework code that can properly create the data and partition it at run–time based on these values. As a result, these two values are supplied in constructors at run–time, as shown in Figure 3.8(b).

When constructing a pattern template from a design pattern, it is important to balance generality against usefulness. Adding more template parameters makes a pattern template more general by allowing more members of the solution family to be specified. However, these additional parameters can make the complete specification of the template more difficult. In contrast, though, fewer template parameters require more of the basic pattern structure to be fixed, which can limit the usefulness of the template by restricting the range of problems to which it can be applied. These considerations have to be evaluated on a pattern–by–pattern basis.

3.4.2 Creating Frameworks

After creating a pattern template for a design pattern, the framework implementing that pattern structure must be created. This task will be the most difficult and time–consuming. Much like pattern templates, frameworks must balance simplicity against generality. A framework must be simple enough to use that a programmer will choose to use it over writing the complete application. A framework must also provide sufficient flexibility that its structure can be used across a broad range of applications.

The process of creating and evolving an object–oriented framework has been captured in a pattern language [88], like the pattern language for parallel programming from Section 2.2.3. The initial creation of a framework is guided by generalizing across several example problems. For the parallel structures provided by the framework generated by the pattern templates, this means examining several problems that use the parallel structure and identifying the common structure and abstractions. Subsequent patterns in this pattern language cover the organization and decomposition of framework classes, in particular concentrating on providing flexible ways of introducing application–specific functionality into the framework.

Frameworks for the PDP process have three other concerns that must be addressed as well. The first concern is the choice of programming language. The process itself is independent of the programming language that will be used to develop applications. Different languages have different facilities for concurrency and parallelism. Languages with no facilities will need support libraries to augment their capabilities. In using Java, CO_2P_3S can use the existing thread facilities to write parallel programs. A system built on C++ will need a library to provide parallelism (such as pthreads or MPI).

The second concern is the type of architecture that the framework will support. CO_2P_3S is targeted at shared memory multiprocessors, and takes this into consideration in the frameworks it generates. For example, the data for the Mesh framework is shared among all of the threads in the program (via the `BoundedMorphogenPairArray` object). A distributed memory system will need to use a distributed shared memory system or need to generate message passing code and distribute data over the processors.

The final concern is the means used to support framework composition. This issue must be considered as even small parallel programs may consist of several patterns, which requires that the generated frameworks must be composed into a complete application. Section 2.3.2 listed the potential problems and possible solutions to this problem. As frameworks become more prevalent, we expect that both lists will grow, as we encounter more problems but create new methods of dealing with them. Thus, the means of framework composition is not dictated by the PDP process, but is instead left to the tool developer.

3.5 Summary

This chapter examined the PDP process, a pattern–based approach to the development of parallel programs. From a programmer perspective, the process consists of five steps that support three layers of abstraction. The layers support the complete task of developing and tuning a parallel program. The complete process was demonstrated using a reaction–diffusion texture generation program, which was parallelized using the Mesh parallel design pattern template in CO_2P_3S.

The PDP process is also a target for parallel systems developers. In creating such a system, there are several issues that must be addressed regarding the creation of design pattern templates and the frameworks generated for the templates. This chapter also examined these issues.

Chapter 4

CO_2P_3S: Correct Object–Oriented Pattern-based Parallel Programming System

The previous chapter detailed the PDP process, a new methodology for creating design–pattern–based parallel programs. To make the process more concrete, it was illustrated using a reaction–diffusion example implemented with the CO_2P_3S parallel programming system. In the example, CO_2P_3S was used to highlight the use of design pattern templates to specify the structure of the program, the use of pattern template parameters to refine that structure, and the generation of framework code for this structure. This framework code provides a correct implementation of the selected parallel structure, which is augmented with application–specific hook methods to create a complete parallel program.

While the PDP process outlines the basic development methodology, it does not fully specify how a tool should support all of the abstractions provided by the three layers of the process. A tool can address other aspects of correctness in framework programming, and must consider usability issues. By leaving some of these details out of the process, future tools can explore new ideas for presenting these abstractions and supporting framework programming.

This chapter covers the CO_2P_3S parallel programming system in more detail, showing how the tool addresses the different aspects of the PDP process and general program development. Again, CO_2P_3S is an example of a tool providing the abstractions in the PDP process. Other tools may choose different mechanisms for presenting the abstractions to the user.

Section 4.1 gives a more complete overview of the CO_2P_3S system, including the program editing, compilation, and execution facilities. It uses the reaction–diffusion example to show how the tool implements some of the features of the PDP process. The patterns supported by CO_2P_3S are briefly described in Section 4.2. Section 4.3 puts CO_2P_3S into perspective with respect to related research. The system is evaluated using the 13 characteristics of ideal pattern–based parallel programming systems to show that it meets more of them than other systems. Finally, some of the abstractions in CO_2P_3S are correlated with research in object–oriented frameworks and modeling languages.

4.1 CO_2P_3S Overview

This section details the features in the graphical user interface for CO_2P_3S. We concentrate on features that enhance framework usability beyond that already discussed in Section 3.2.3. There, framework usability was addressed by encapsulation and conditional code generation. Encapsulation hides the details of the framework so the programmer can concentrate on only those portions of the framework that are needed to build an application, and prevents users from introducing errors into the structural code. Conditional code generation allows the user to refine the structure of the supported patterns using template parameters. The generated code also includes concrete classes needed to insert application–specific functionality, with appropriate stubs for the hook methods used by the framework. This section shows how the CO_2P_3S user interface can further improve framework usability and reduce the probability of user error.

Two screenshots from CO_2P_3S, showing different panes on the right side, are shown in Figure 4.1. We will examine each pane in the interface in the following subsections. The compilation and execution facilities in CO_2P_3S are also examined.

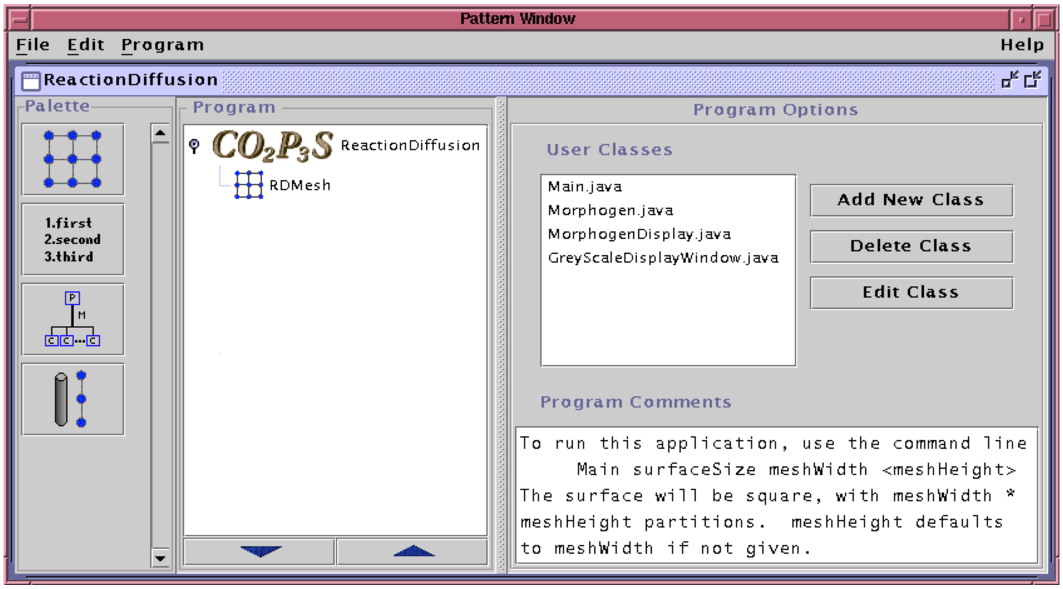
4.1.1 Pattern Palette

The Pattern Palette, shown in Figure 4.2, presents the user with the pattern templates supported by CO_2P_3S. In the figure, the templates, from top to bottom, are

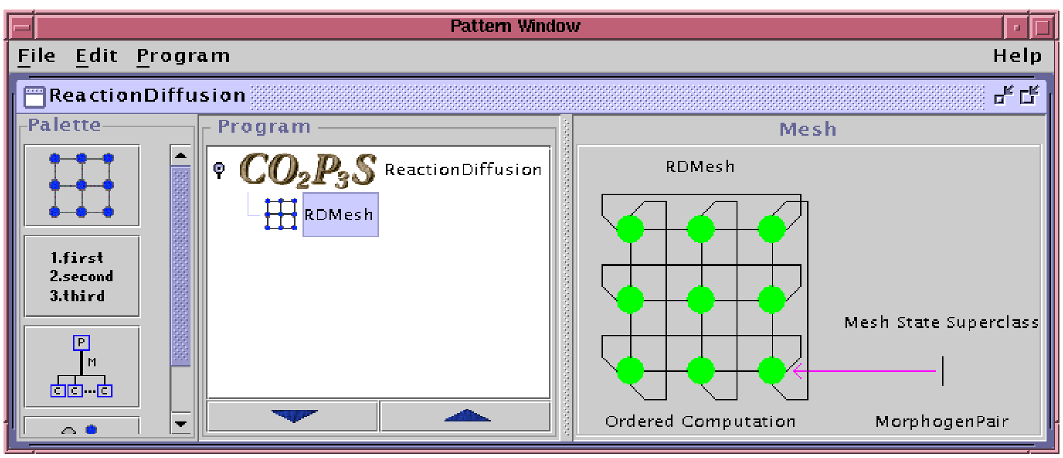
1. the Two–Dimensional Mesh,
2. the Phases,
3. the Distributor, and
4. the Pipeline.

These patterns will be covered in more detail in Section 4.2.

The number of supported patterns at the time of this dissertation is small, but was driven by the needs of the example applications that have been created with CO_2P_3S, which are discussed in Chapter 5. Similarly, the capabilities of the frameworks for the current pattern templates were driven by the needs of these programs. The development philosophy behind this work, much like



(a) A screenshot of CO_2P_3S with the Program Options pane on the right.



(b) A screenshot of CO_2P_3S with the Pattern pane on the right, from Figure 3.3.

Figure 4.1: Screenshots from CO_2P_3S.

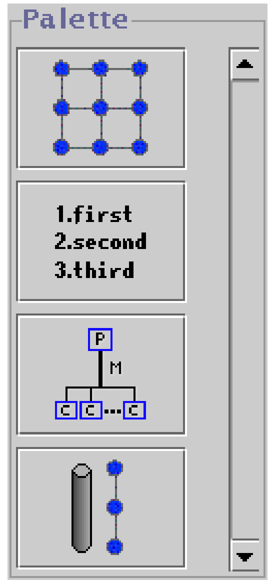


Figure 4.2: The Pattern Palette, for selecting design pattern templates.

that for Extreme Programming [102], was to create what we needed when we needed it and little more. As more applications are created, the set of patterns and the capabilities of the generated frameworks will grow.

A user includes an instance of a template in an application program by clicking on it. The new instance will appear in the Program pane (Section 4.1.2). In addition, the user can access pattern template documentation through a popup menu on the templates in the palette.

The most interesting aspect of the pattern palette is the use of the reflective capabilities of Java to add the templates. The list of template names to be included in the palette is part of a configuration file (`.copsrc`, in the user's home directory). The name corresponds to a class representing the template, which is instantiated by the user interface. One of the responsibilities of this class is to provide access to its help and to provide a small graphical representation of itself for the palette. By using the reflective capabilities, it is possible for users to easily customize the set of available pattern templates. More importantly for future research is that this approach allows new templates to be integrated into the interface without requiring any changes to existing code in the system. To add a template (assuming that the classes implementing it are available), the user need only add the name to the list of templates in the configuration file.

4.1.2 Program Pane

The Program pane shows the current set of pattern templates that the user has selected for the parallel structure of a program. It also shows the program name and provides access to the Program Options pane (Section 4.1.3). Figure 4.3

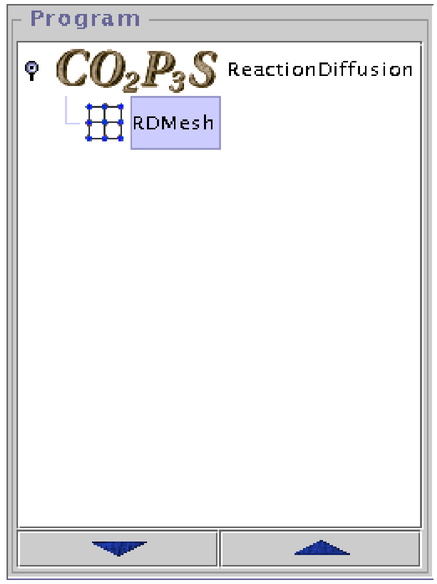


Figure 4.3: The Program pane, showing the set of selected design pattern templates for a program.

shows the pane for the reaction–diffusion application.

The topmost item in the list, with the CO_2P_3S logo, represents the complete program. The text beside the logo is the name of the program, entered when the program is created. Selecting this item causes the Program Options pane to appear to the right, resulting in the screenshot in Figure 4.1(a).

Underneath the topmost item is a list of the pattern templates that have been selected for this program. Each pattern is indicated by a small version of its icon and the name of the template. The name of the template is taken from one of the class names in the parameters for that template. Since these names must be unique for a given program, the user can distinguish between different instances of the same template. In the Mesh pattern, the template name is the name of the mesh class, which was set to `RDMesh` (Section 3.2.2). If the user has not yet entered this class name, the name of the template is used. Clicking on one of the templates in the list in the Program pane causes the pane to the right to become an instance of the Pattern pane (Section 4.1.4) displaying the selected template, yielding the screenshot from Figure 4.1(b).

4.1.3 Program Options Pane

The Program Options pane allows the user to supply options that apply to the entire program. This pane is shown in Figure 4.4.

The top section of the Program Options pane allows the user to manage any additional user classes that are used by the program but are not generated with the frameworks for the pattern templates. CO_2P_3S supports the inclusion of these auxiliary classes because users should be able to reuse their existing

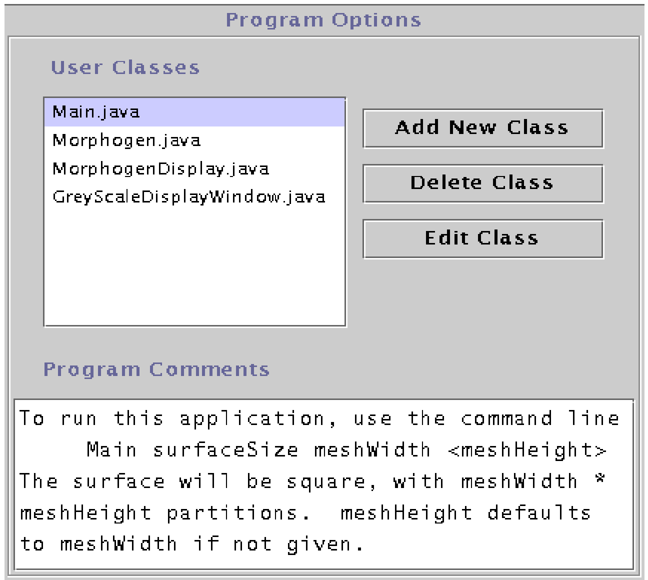


Figure 4.4: The Program Options pane, showing the list of external classes and program comments.

code where possible. Specifically, since the hook method implementations for the generated frameworks are sequential code, we should make provisions for creating additional classes or including existing classes, possibly from a sequential implementation of the program. These classes can be added, edited (with the user's choice of editor), and removed from a program using the three buttons in the pane.

The bottom section of the Program Options pane is a text field that allows the user to add comments to the program. These comments may document the command line options, as in this example, or provide additional design information for program maintenance.

4.1.4 Pattern Pane

The Pattern pane shows the current state of the design pattern template that is currently selected in the Program pane. This pane is used to manipulate the template. The Pattern pane for the instance of the Mesh template is given in Figure 4.5.

The graphical representation of the pattern template indicates the current state of the template so that it is clear to the user at a glance. The state of a template includes the values of its parameters and the layer at which the template was last manipulated. In Figure 4.5, the parameters for this particular mesh are apparent. The mesh is fully–toroidal (indicated by the arcs), four–point (as the centre node has only four nodes), and ordered. The class names are `RDMesh` for the mesh and `MorphogenPair` for the mesh elements. The green nodes indicate that this pattern has been used to generate frame-

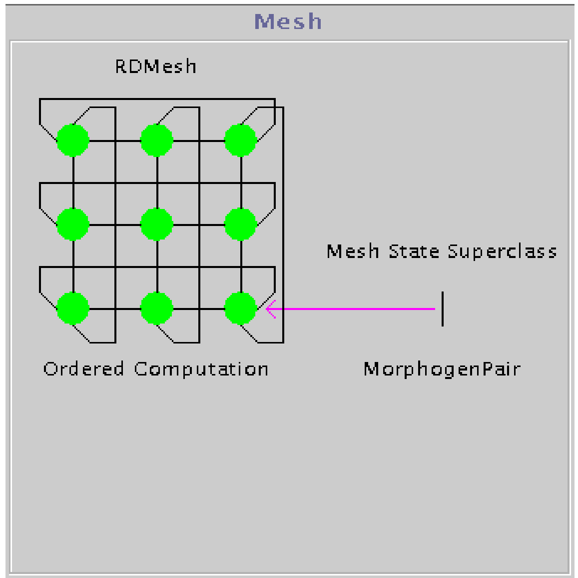


Figure 4.5: The Pattern pane, showing one of the design pattern templates in a program.

work code at the Patterns Layer. If framework code has not been generated or is out of date (because a template parameter has changed), the nodes will be red. At a glance, the user can determine the state of the selected template.

The parameters for the templates can be modified through popup menus in this pane. Each template can add menu items specific to itself for its parameters.

The Pattern pane also performs some error checking for the pattern templates. For instance, it verifies that there will not be any class name conflicts between the different templates. Each template must provide a means of determining if a given class name will conflict with any of the classes that may be generated for its framework, including those classes that are not visible to the user at the Patterns Layer. The interface does not permit any conflicts. The template parameters are also split into two categories: those that have legitimate default values that can be used when a framework is generated, and those that must be supplied by the programmer. The boundary conditions of the Mesh template are an example of the former; the default conditions are non–toroidal. The class name parameters are examples of the latter. They must be supplied before framework code is generated for a pattern template. The user interface can enforce this constraint.

The Pattern pane also allows the user to view the classes that provide hook methods for the template and insert code into these classes. Figure 4.6 is an example of the template viewer in CO_2P_3S, which provides access to the `MorphogenPair` class code for this example. This class holds default stubs for all of the hook methods provided in the framework. Application code that has already been entered for the hook methods is also visible. The viewer is a modified HTML viewer, where the links in the document are places that the user can insert application code. The most common links are method and constructor signatures, where the user enters code for the body. In addition, we include links to allow the user to enter code in other sections of the class. One such link is at the top of the code for adding import statements, which we used to import the `java.util.Random` class. There is also a link after the stubs for the hook methods that allows other methods and instance variables to be added to this class, that is not shown in Figure 4.6.

The viewer has two other useful features. First, any code displayed in the viewer is the output of a code formatter, so that the results are consistently indented and readable. This is particularly important since the user does not have complete control over the indenting of the application code. The second feature is the `Show Line Numbers` check box in the bottom right corner. When this is checked, the line numbers appear at the left side of the source code. This makes is easy for the user to quickly track down any compile errors, which are referenced by source code line number.

When the user clicks on a link in the viewer, the code editor dialog appears. This dialog is shown in Figure 4.7. The editor shows the signature of the method or constructor being edited, with a text area for entering application code below it. If code has already been entered for the method, it will appear in the dialog. If the code changes are accepted by the user, the code in the viewer will update.

The combination of the viewer and code editor dialogs reduces the probability of programming errors in two ways. First, the user does not have to type in the signatures. The class displayed in the viewer is part of the generated framework, which includes stubs for the needed hook methods. Second, the user cannot accidently change the signatures. These signatures are important as they are part of the interface used by the abstract structural classes to invoke the application–specific code provided in the concrete classes. Any changes to the signatures can cause the concrete classes to become incompatible and the framework may no longer work correctly. The combination of the viewer and the code editor prevent the user from modifying the method signatures, which prevents this type of error.

4.1.5 Compile and Run Dialogs

Any good programming system must not only support the development of programs, but also support compilation and execution. CO_2P_3S is no different. The Compile dialog for CO_2P_3S is shown in Figure 4.8, and the Run dialog is shown in Figure 4.9.

The Compile dialog allows common compilation options to be selected using the checkboxes. Additional flags for the compiler can be specified in the `Flags` text field. The `Compile` button compiles the complete program. The output of the compile appears in the text area at the top of the dialog. The

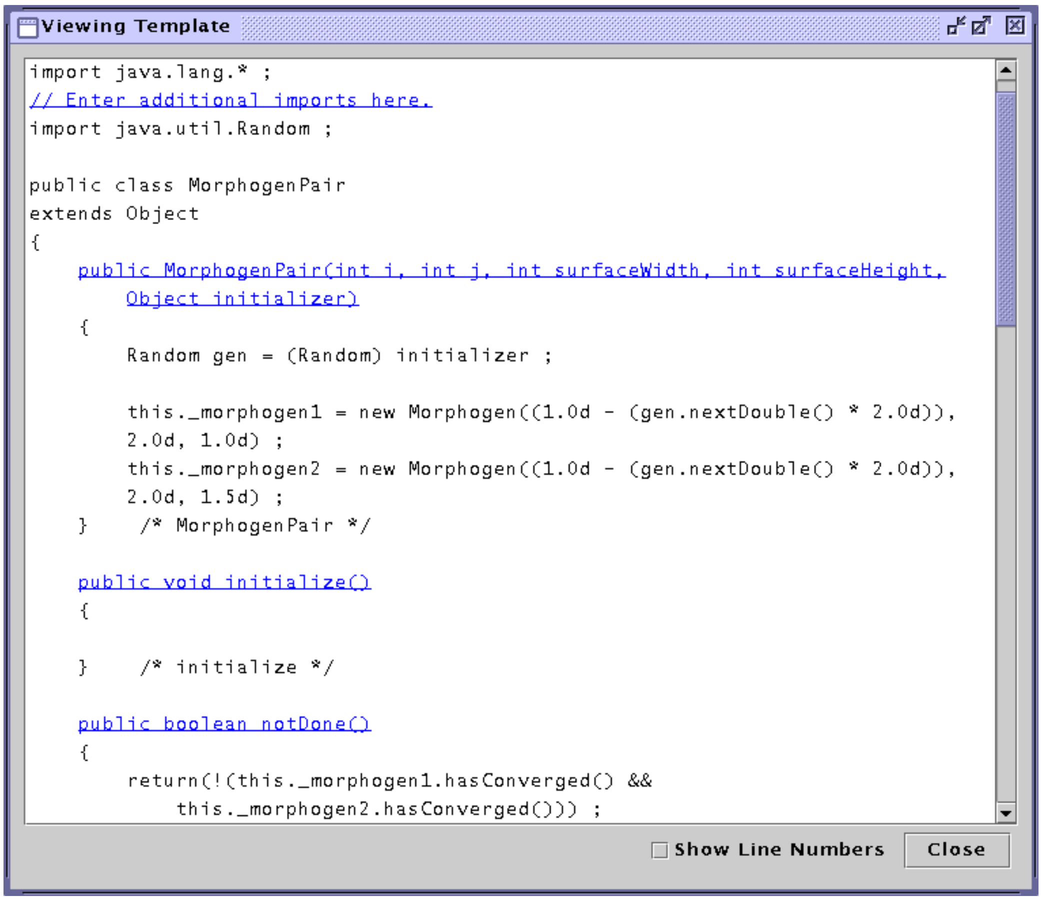


Figure 4.6: The Viewing Template dialog for viewing and entering hook method code into a generated framework. Underlined text are hypertext links to sections of the class that can be edited.

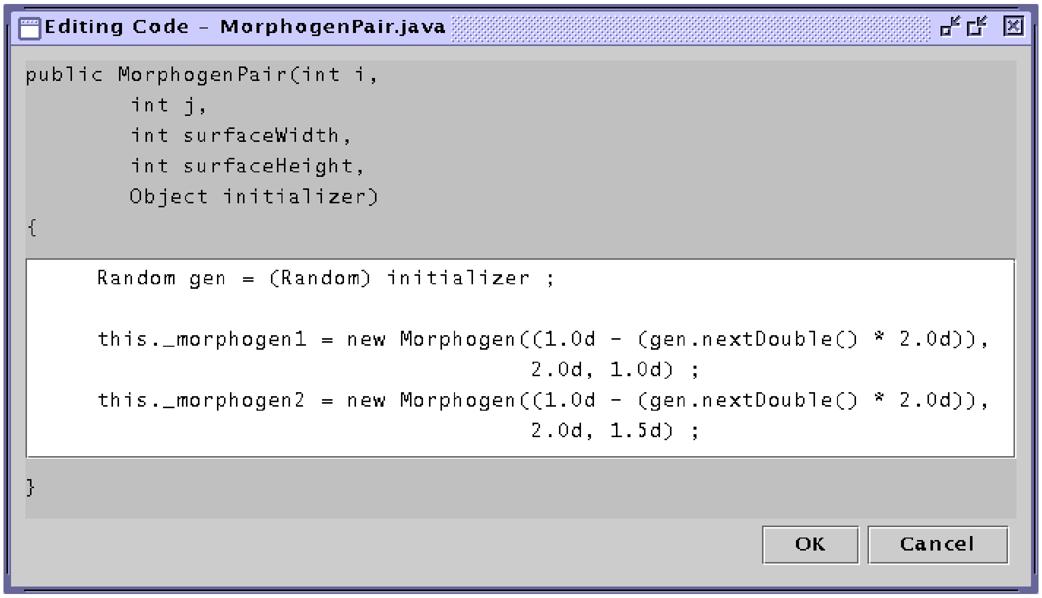


Figure 4.7: The Editing Code dialog for entering hook method code.

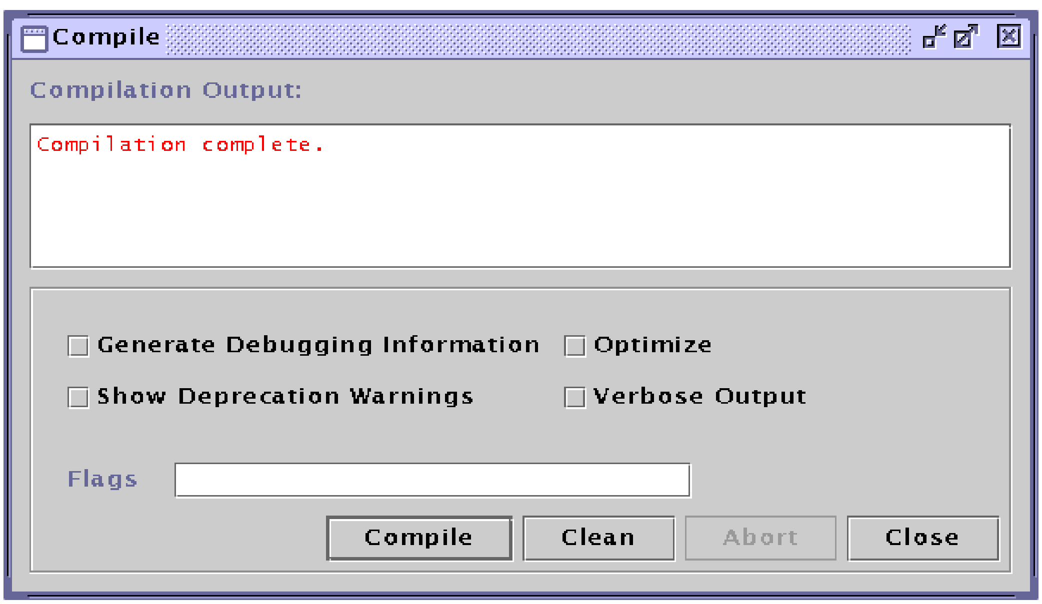


Figure 4.8: The Compile dialog, for compiling programs.

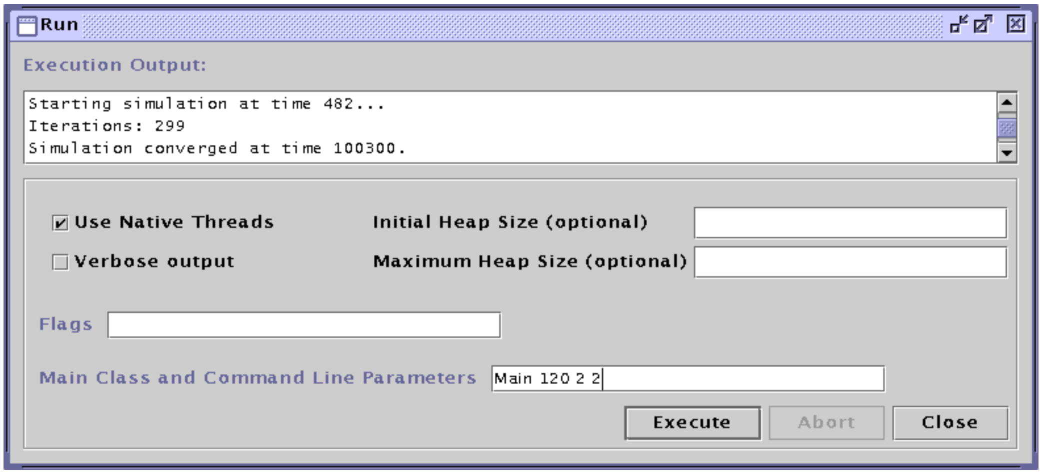


Figure 4.9: The Run dialog, for executing programs.

`Clean` button removes all Java `.class` files for an application. Finally, an executing compilation can be stopped with the `Abort` button.

The Run dialog is similar to the Compile dialog. Again, program output appears in the large text area at the top of the dialog. The most common run–time options appear as check boxes and the heap size text fields. Extra options can be entered in the `Flags` text field. The mainline class and command line parameters are entered in the bottom text field. Finally, there are buttons for executing the program and aborting the current execution.

Because CO_2P_3S is targeted at shared memory multiprocessors, there is no facility for configuring which computers will take part in a computation. Tools that will execute programs on networks of workstations or clusters of distributed memory machines will need to include this configuration data. This can be part of the Run dialog or may be part of the general tool configuration.

4.2 Parallel Design Patterns Supported by CO_2P_3S

A crucial aspect of any design–pattern–based system is the set of patterns it supports. This will determine the range of problems that can be efficiently implemented with the tool.

This section gives more details on the parallel design pattern templates supported by CO_2P_3S at this time. These templates are:

- Two–Dimensional Mesh. This template supports iterative computations for a regular, rectangular two–dimensional set of data, where this data is decomposed into a set of regular, rectangular partitions that are distributed over a set of threads. This template is discussed here only briefly as it was covered in detail in Chapter 3.

- Distributor. This template supports data–parallel style computations by forwarding methods from a parent object to a fixed number of child objects, all executing in parallel.
- Phases. This template supports the creation of phased computations by invoking an ordered sequence of methods on a Facade [37] object.
- Pipeline. This template supports pipeline computations. These computations consist of a series of independent stages that process a stream of input items, where each stage refines its input and forwards its output to the next stage. Since the stages are independent, each can be refining a different item in parallel with the other stages.

The details of these templates are specific to CO_2P_3S. Other systems may choose different templates, or may choose different alternatives for the above templates (for example, a different Mesh template may include support for irregular meshes or may have a different set of parameters).

The descriptions of each template describes what it does and how to use it. Fully documenting a template should include several motivational sections that are part of normal pattern documentation, such as Intent, Motivation, and Applicability. Appendix B gives the format for our complete template description, and Appendix C is an example of the documentation using the Two–Dimensional Mesh. This documentation is intended for users at the Patterns Layer, and so is intended to provide information on how to use the template to create a program rather than on how the generated frameworks are implemented.

Note that we devote more attention to the Pipeline than the other templates in this section. We have a new formulation of the Pipeline that addresses some of the weaknesses in more traditional versions. This new formulation is based on a combination of the State pattern [37], a separation between the concurrency and the object structure, and a careful examination of the necessary ordering between items as they progress through the pipe. These are used to address the load balancing problems that cause many programmers to avoid this pattern in their programs.

4.2.1 Distributor

The Distributor pattern template provides a data–parallel style of computation. The pattern consists of a parent object that encapsulates a fixed set of child objects. Methods are invoked on a parent object. Some of these methods, as indicated by the user, will invoke a method with the same name on a fixed number of child objects in parallel, each child operating independently. Optionally, one–dimensional array arguments can be automatically distributed across the children using one of four supported distribution strategies.

The crucial property of this pattern template is the encapsulation of the child objects. The children should not be visible outside of the parent object,

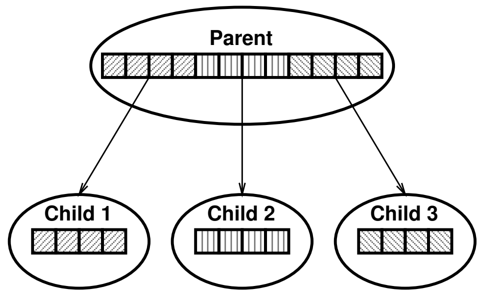


Figure 4.10: Distributing an array over a set of child objects.

so all methods must be invoked on the parent. In addition, to enforce their independence, the children should not have references to one another. Thus, all references to the child objects must be done indirectly via the parent object, which can control access to them.

This template is particularly effective when state can be split into independent parts and distributed over the child objects. Methods that manipulate this state can be executed in parallel where possible. The most obvious example is an array that can be split into subarrays and distributed over the child objects, as in Figure 4.10. Operations on the data in the array are invoked on the parent. If this operation has been identified as being a parallel method, the operation is forwarded to each child, which performs the operation on its portion of the array in a data parallel fashion.

Some of these methods may not be able to execute in parallel though. The granularity of the method may be too small to warrant parallel execution or the method may have data dependencies that prevent concurrency. For the Distributor, this does not present a problem; the user can add sequential methods to both the parent and child classes. These methods can manipulate the state of the children or can operate on state in the parent that could not be split into independent pieces. This provides the user the freedom to determine where parallelism can be best applied inside the object structure for the Distributor.

The parameters for the Distributor design pattern template are:

1. The class name for the parent class. The class name for the child class will have "Child" appended to the end of this class name.

2. A list of methods that will be executed in parallel on the child objects. This list has the following elements:

   (a) The return type of the child implementation of the method. The parent returns an array of these objects, one per child, unless this type is `void`.

   (b) The method name.

(c) The arguments to the parent implementation of the method. Optionally, one–dimensional array arguments can have a distribution scheme applied to them. The distributions are:

Pass through distribution Each child gets a reference to the whole array (Figure 4.11(a)). This is the default distribution.

Block distribution The array is split into n contiguous subarrays, where n is the number of children. Each child gets a unique subarray (Figure 4.11(b)).

Striped distribution Each child i will receive an array containing the elements at indices $(i,\ i+n,\ i+2n,\ \ldots)$, where n is the number of children (Figure 4.11(c)).

Neighbour distribution The ith child gets a 2–element array consisting of elements i and $i+1$ from the argument array (Figure 4.11(d)). This distribution scheme is used in the parallel sorting example discussed in Section 5.1.

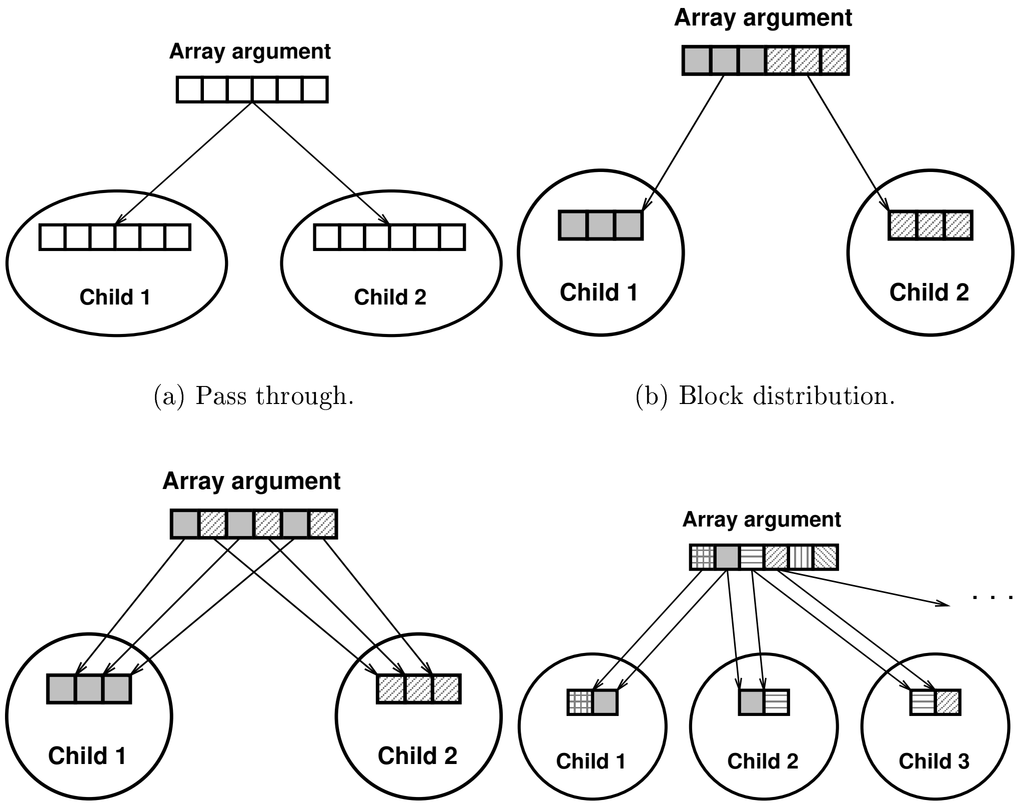
All other arguments are passed through. The child implementation of the method will have the same arguments as the parent, passing an array of elements even if only one element is actually passed.

The number of child objects is a parameter to the constructor in the framework generated for this template so that it can be set at run–time.

The Distributor template, as it appears in CO_2P_3S, is shown in Figure 4.12. The left side of the Pattern pane graphically depicts the object structure of this template, with the parent and child class names (`DistExample` and `DistExampleChild` respectively). The `Method List` on the right side of the pane shows the current set of methods that will be executed in parallel (in this case, the single method `initialize()`). The red nodes and red rectangle around the list of parallel methods indicate that framework code has not been generated for this template.

The dialog for entering parallel methods for the Distributor template is shown in Figure 4.13. The dialog provides the means of specifying all of the elements of the method list (the second template parameter) in a controlled manner. The distribution scheme can be applied only if the corresponding argument is a one–dimensional array, indicated by the presence of "`[]`" at the end of either the argument type or the argument name. The figure shows the drop–down menu for the available schemes that the user can select. For arguments that are not arrays, the only available option is pass through.

The structure of the framework generated for the Distributor design pattern template is shown in Figure 4.14. The user's view of the framework consists of the two classes shaded in grey, the concrete class for the parent and the concrete class for the child. The programmer instantiates the framework by creating an instance of the parent class. This creates the child objects, the number of which is an argument to the constructor. The hook methods in the framework are the child implementations of the parallel methods, which



(a) Pass through. (b) Block distribution.

(c) Striped distribution. (d) Neighbour distribution.

Figure 4.11: Distribution schemes that can be applied to one–dimensional array arguments in the Distributor pattern template.

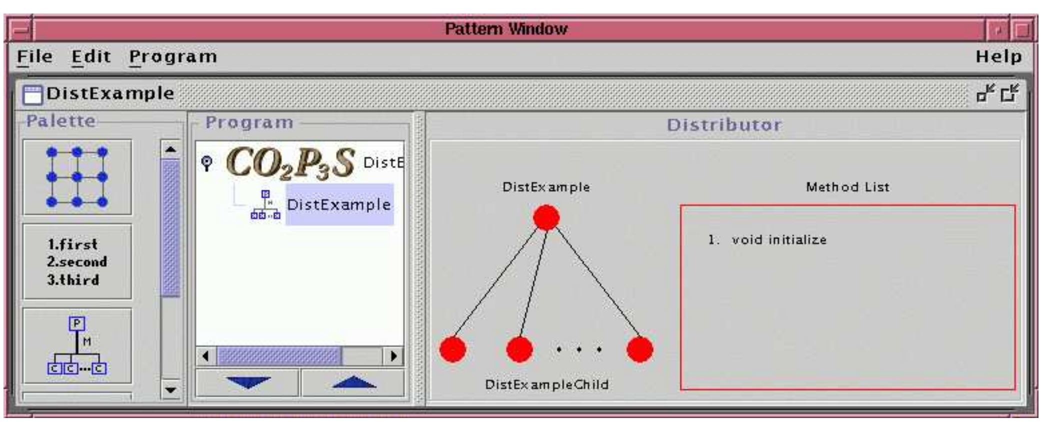


Figure 4.12: A screenshot of CO_2P_3S showing the Distributor design pattern template.

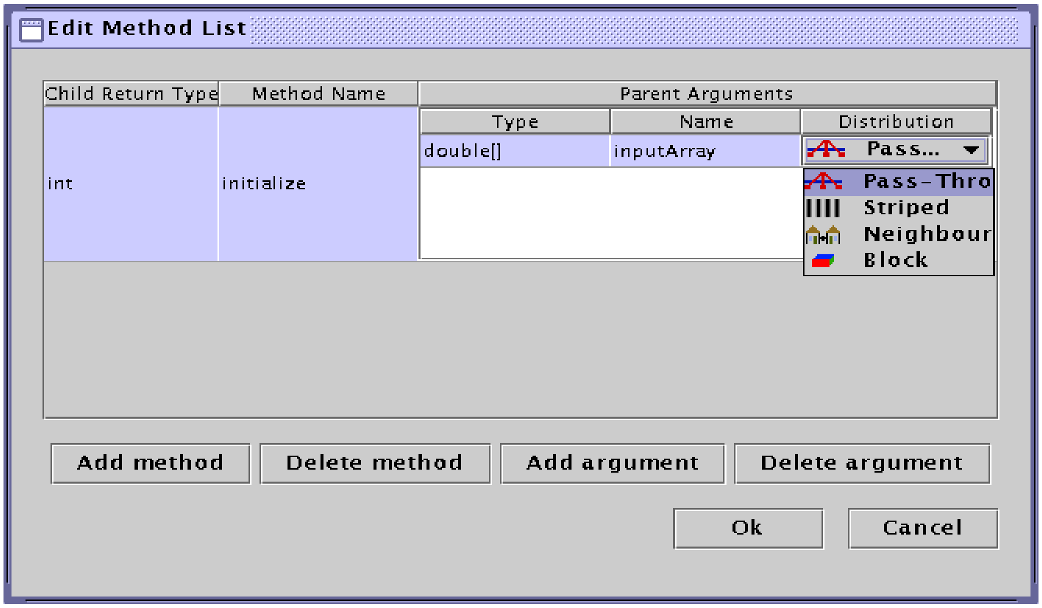


Figure 4.13: The dialog for entering parallel methods for the Distributor template, including a list of the distribution options for one–dimensional array arguments.

the user enters through the viewer from Section 4.1.4. The strategy objects implement the various distribution schemes, except for pass through.

When using this framework, all methods are invoked on the parent object. The abstract superclass provides an implementation of each of the parallel methods listed in the pattern parameters, which does the following:

1. The correct arguments for the child method are assembled. This may require the use of the strategy objects implementing the distribution schemes.

2. A set of threads is created, one per child, to execute the method on the

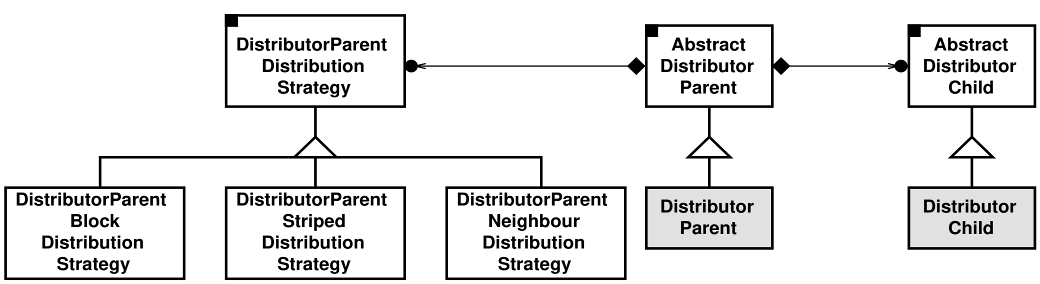


Figure 4.14: The class structure of the framework for the Distributor template. The classes that are visible to the user at the Patterns Layer are shaded in grey.

children. They are augmented Java threads that use reflection to find the correct method (based on a method name and the best match of the supplied parameters) and provide return values when they are joined [65].

3. The parent waits for the threads to finish. The results, if any, are put into a result array. These results are returned to the caller.

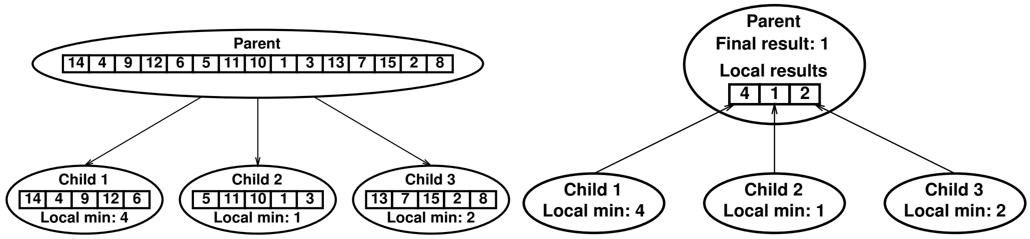
The granularity of the child methods must outweigh the cost of creating the threads. There are times when a method does not warrant parallel execution, such as distributing data across the children in an existing instance of the Distributor framework. In this case, it is possible to add additional sequential methods to both the parent and child classes (the latter of which must be invoked indirectly through the parent). These added sequential methods are not part of the parallel structure of the framework, so no code is generated for them and no hook methods are created. Instead, the Distributor template provides viewers for both the concrete parent and concrete child classes, which can be used to add new methods and constructors.

Additional sequential methods can also be used in cases where there is state information that logically belongs in the parent object but cannot be distributed across the children, possibly because of dependencies in the data. This state can be stored in the parent and manipulated using methods added to the parent class.

Two example uses of sequential methods in the Distributor framework are the control of parallelism and the implementation of simple parallel reductions. In the first example, consider a method `arrayOperation()` that distributes an array argument over the children and performs some operation on it. The granularity of this method depends on the size of the input array. If the array is too small, then the operation should be done sequentially by the parent. This can be implemented by creating another method in the parent object that first checks the length of the input array and, if the array is long enough, calls the parallel method `arrayOperation()` with the array. Otherwise, the operation is performed directly on the input array.

The second example, parallel reduction, processes a collection of data and reduces it to a single value. Finding the smallest number in an array is a good example. Reductions can be implemented using a tree–like approach. The input data is spread over a set of processors, each of which computes the reduction for its portion of the data. These partial results are combined and the global value is determined.[1] Figure 4.15 shows how to find the smallest number using a tree of depth one. The basic Distributor framework can do part of the reduction but not all of it. The return value of a parallel method in the Distributor framework is an array of results, one per child, where the final result of a reduction should be the reduced value of this array. To implement

[1]The operation must be commutative for a parallel reduction to work properly.



(a) The parent distributes the array over the children, each of which finds the minimum element of their portion of the array.

(b) The parent takes the partial results from each child and finds the global minimum.

Figure 4.15: Finding the smallest element in an array using a tree–based parallel reduction.

the reduction properly, the user can add a sequential method to the parent object. This method invokes the parallel reduction, allowing the children to find the local result, and then finds the global result by reducing the local results.

The other obvious implementation for this framework is to use a fixed pool of threads, which execute requests for the parent object. Another option is to use an implementation of a `forall` construct, such as [82], which reduces the cost of thread creation and provides better support for data parallelism. The primary benefit to the current implementation is that it is simple to build and understand, and hence easier to modify at lower layers. Also, the applications that we address typically execute for a sufficiently long time that any overhead introduced by the creation of threads and reflection represent an insignificant fraction of the total run time. The sorting example in Section 5.1, which uses two instances of this template, takes several minutes to execute, which absorbs the overhead of thread creation and reflective method lookup.

4.2.2 Phases

Both the Two–Dimensional Mesh and Distributor patterns concentrate on creating and using concurrency for executing problems in parallel. Clearly this is a crucial aspect for parallel programming. Equally important is synchronization. The concurrency must be controlled in some cases to ensure that certain operations are not performed by multiple processes simultaneously. Without this control, these operations could conflict and result in errors.

Another application of synchronization is the creation of *phased algorithms*, in which the necessary parallelism varies as the algorithm progresses. This is in direct contrast to the assumption of early skeleton and framework research, which assumed that a single structure was all that was necessary in a given application (Section 2.1.1, the *hierarchy* and *independence* characteristics).

For example, the sorting example is Section 5.1 has a clearly defined set of phases that must be completed in a specific order, with a subset that are expensive enough to warrant the use of parallel pattern templates.

In creating phased algorithms, there must be a mechanism for passing data generated in one phase to other phases that need it. However, this data may be intermediate results that should not be exposed outside of the phases that create or consume it. Encapsulating this data should be part of creating a phased algorithm.

The Phases design pattern template provides a specialization of the Facade design pattern [37] that supports an extendible means for creating phased algorithms. This facade invokes an ordered set of methods, respecting the necessary ordering so that a phase is only invoked when the previous phases have completed. The facade can also encapsulate temporary results which can then be supplied to subsequent phases.

In its simplest form, the Phases pattern is an ordered sequence of method invocations, where each method represents a different phase of the overall computation. This assumes that all parallel activities in a given phase are finished at the end of each method. In more general cases, one can imagine a directed acyclic graph of method dependencies that allow independent phases to execute in parallel. While the Phases template uses the former implementation in its generated framework (as the concurrency in our parallel frameworks is finished at the end of the computation), the latter implementation can be constructed by assigning threads to execute each phase and adding a barrier or rendezvous to ensure that a given phase does not start executing until the ones it depends on have finished.

The parameters for the Phases template are:

1. The name of the Phases class.

2. An ordered list of the names of the methods that will execute each phase. These methods have no parameters and return no results.

The Phases design pattern template in CO_2P_3S is shown in Figure 4.16. Graphically, it simply lists the different phases. The Method List Editor, which allows the user to enter the phases, is also shown in the figure.

The structure of the framework generated for the Phases template is shown in Figure 4.17. The user's view of the framework consists of the concrete class shaded in grey, which contains the stubs for the methods listed in the second template parameter. In addition, extra state and constructors can be entered into this class. To use the Phases framework, the user instantiates the `Phases` class and invokes the `executeSequence()` method on this object.

The key to this pattern template is that each phase can be implemented separately. This can be done using another parallel design pattern template or using sequential code. This template provides no concurrency or parallelism on its own.

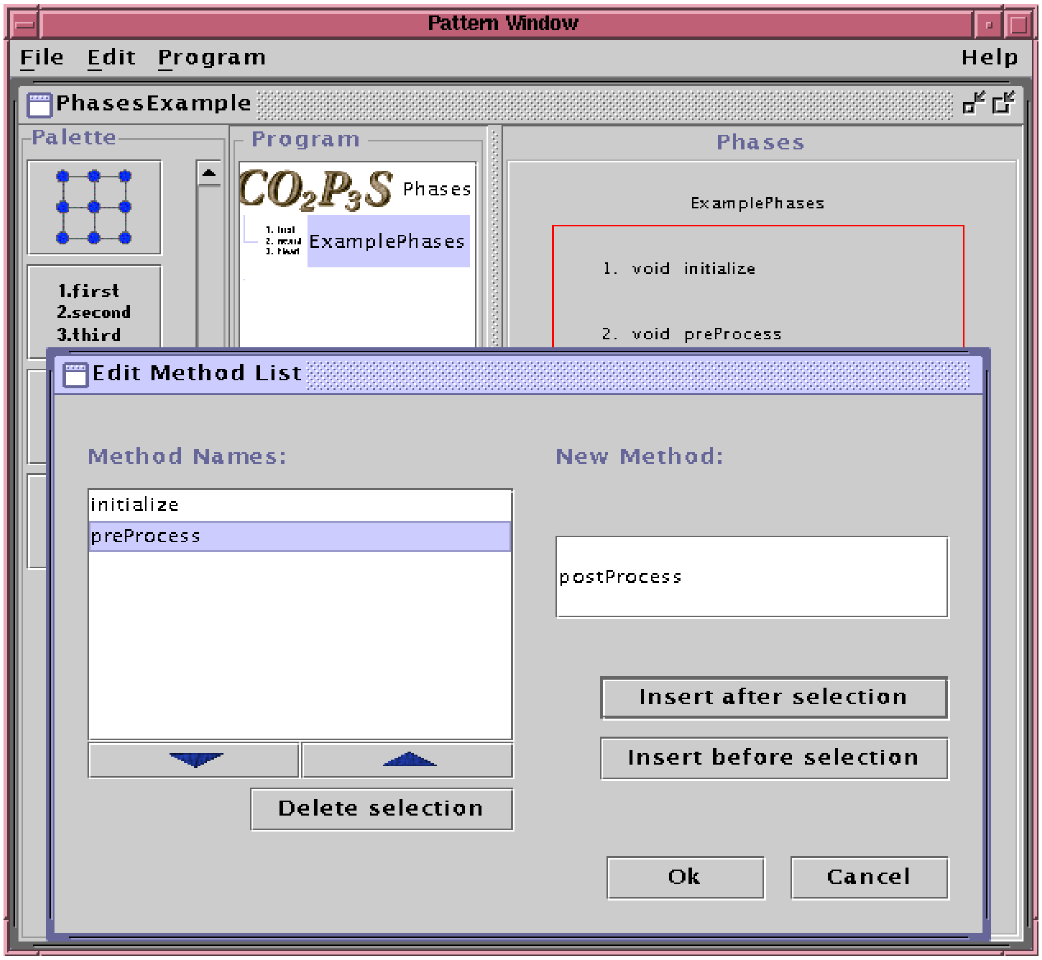


Figure 4.16: A screenshot of CO_2P_3S showing the Phases design pattern template.

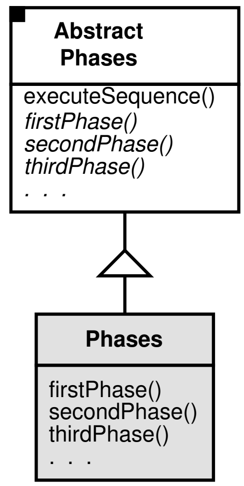


Figure 4.17: The class structure of the framework for the Phases template. The classes that are visible to the user at the Patterns Layer are shaded in grey.

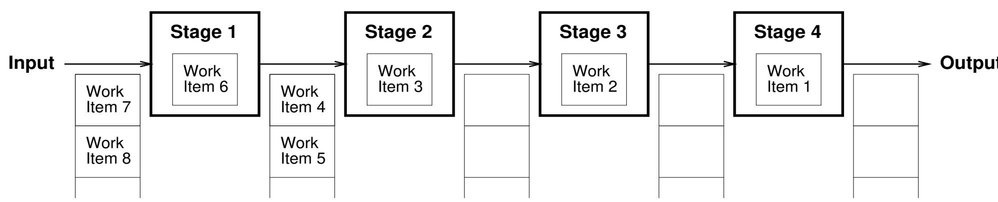


Figure 4.18: An example of a pipeline.

4.2.3 Pipeline

Pipelines are ubiquitous in both computing applications and real–life situations. Pipelines are used by all modern processors to speed up the decoding and execution of instructions. Factories use assembly lines to speed up their manufacturing processes. A pipeline structure is a simple strategy for improving application performance.

A pipeline is shown in Figure 4.18. It is an ordered sequence of stages, where each stage refines its input and sends its output to the next stage. This transfer is usually handled by some form of buffer, shown between the stages in the figure. The computation of each stage, however, is independent. The parallelism in this structure comes from the ability to have different stages processing different requests simultaneously. This allows each stage to be assigned to a different processor. For example, in Figure 4.18, each stage is concurrently executing a different work item. This allows an input stream of work items to be processed efficiently. In the ideal case, the speedup obtained from using the pipeline is equal to the number of stages.

Weaknesses of the Pipeline

Achieving the optimal speedup from a pipeline is difficult for several reasons. First, there is a ramp–up and ramp–down time for the pipeline. When the first few requests are being processed, later stages are idle as the pipeline is not full. A similar problem occurs when the pipeline is emptying; early stages are idle as they have exhausted all work. Second, load balancing is critical to obtaining good performance. Any stage that is not in balance with the remaining stages causes performance problems. A stage that does more work than other stages will starve subsequent stages. A stage that does less work than other stages will be idle waiting for work from earlier stages. For instance, from Figure 4.18, assume that the first stage has a shorter execution time than the other stages. It is already executing the sixth work item while the next stage is still on the third item. The first stage will quickly exhaust the input stream and then sit idle as there are no more work items to process. This reduces the number of processors that are effectively working on the problem and thus reduces the performance of the application. This problem can sometimes be addressed by replicating expensive stages, which improves the throughput of those stages. For example, to balance the pipe in Figure 4.18, some of the later stages can be replicated to improve their throughput and balance the entire pipe.

Because of these problems, experienced parallel programmers generally avoid pipeline computations, especially for coarse–grained parallel programs where load balancing problems are exacerbated. To quote Randy Crawford [25],

> If the time spent in each stage of the pipeline is the same, then no stage will starve while waiting for its predecessor to finish. But that's pretty optimistic. IMHO [In my humble opinion], you'd be better off with a 'workpile' model in which you break up the work to be done into small pieces and then dispatch a new thread each time you have more work to do.

However, for novice parallel programmers, the pipeline is a natural solution to some problems. For these users, we would like to provide a template that can be programmed as a pipeline but addresses the load balancing problems. To do this, we show how to transform a pipeline program to a work pile by applying the State pattern [37]. The result is a design pattern template that appears to be a pipeline from the user's perspective but executes using the work pile model to address the balancing problems.

Executing the Pipe as a Work Pile

Most object–oriented pipelines simply replace the stages in Figure 4.18 with active objects. Active objects use their own thread of control to execute any methods that are invoked on them [59]. This basic pipeline can be extended by considering the differences between an earlier stage *pushing* data to later

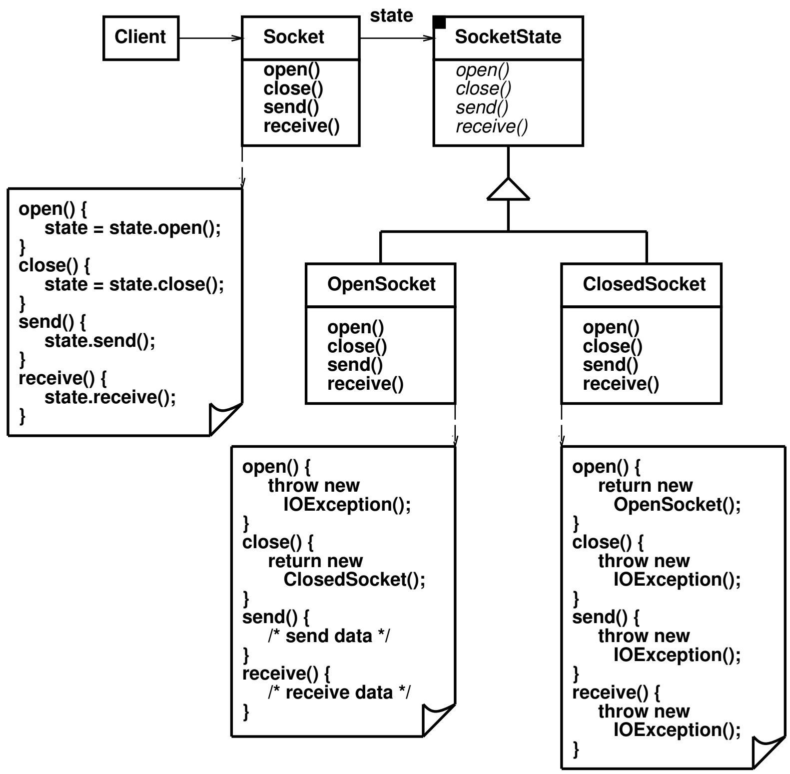


Figure 4.19: An example of the State pattern using a simplified socket.

stages and a later stage *pulling* data from earlier stages [107]. However, the basic transformation from a basic pipe to an object–oriented pipe by mapping stages to objects remains the same.

Instead, let us consider the State pattern [37]. The State pattern represents a set of states that an entity can be in using different classes. The implementation of the methods in each of these classes differs depending on the current state of the entity. For example, consider the simple socket example given in Figure 4.19. This simple socket can be opened or closed, and data can be sent through it or received from it. The two states for the socket, open and closed, are represented by the classes `OpenSocket` and `ClosedSocket`. The `Socket` class hides state transitions from classes using the socket. Each of the state classes implements the four operations in a manner consistent with the current state of the socket. For example, `ClosedSocket` allows a socket to be opened, but throws exceptions if the socket is used or closed again. The benefits to the State pattern are separation of concerns and simplifying the `Socket` class. Each state is a separate class and can be examined and changed individually. The `Socket` class does not need complicated control flow statements to manage the separate states. This makes it easier to add new states.

The first step in the transformation from a pipe to a work pile is to formulate the pipe as an instance of the State pattern. To do so, we can consider the

```
while (not finished) {
    Find a work item w from one of the buffers.
    nextStage = w.transform() ;
    Place nextStage into the appropriate buffer.
} /* while */
```

Figure 4.20: Pseudocode for the actions of each thread in the Work–pile–based pipeline.

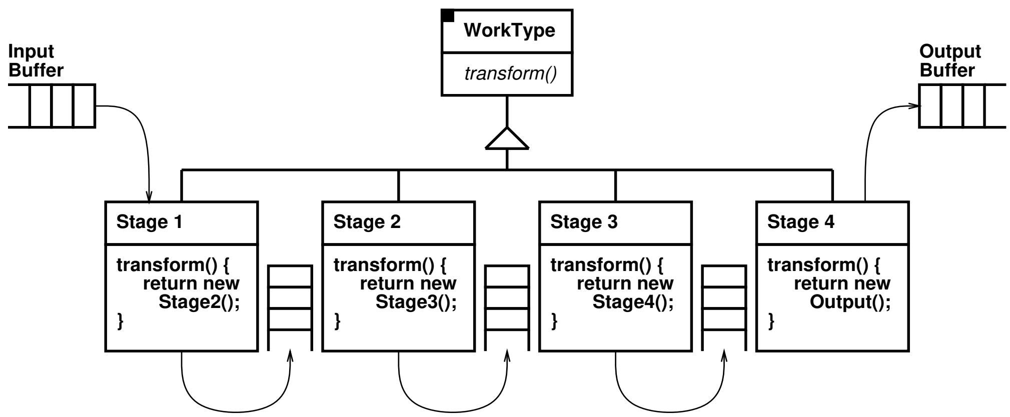


Figure 4.21: An example of the pipeline based on the State design pattern.

pipeline as a sequence of stages that each accept an input object and transform it to an output object of a different type. These transformations can be recast as the transformation from one state to the next, much like the change of an open socket to a closed socket in Figure 4.19. We can use polymorphism to ensure that the same method, `transform()`, is used for the transformation.

The first step defines the transitions but does not address concurrency. We need some mechanism to allow multiple processors to work on the different work states. For this, the second step in the transformation relies on the work pile model. First, we add buffers for holding the results of each stage, much like the buffers in Figure 4.18. After a stage transforms its input object to an output object, the output is placed into the correct buffer. A group of worker threads executes the pseudocode from Figure 4.20, repeatedly searching the buffers, executing the next transformation that is found, and placing the result into the correct buffer. The last buffer is designated the output buffer and contains the results of the final transformation. Therefore, it is not one of the buffers searched by the worker threads. Because the same method implements the transform for any stage (through polymorphism), the threads need no special information to execute any request regardless of the actual stage type. The complete Work–pile–based pipeline is shown in Figure 4.21.

An important aspect of the pipeline that we have not dealt with is ordering.

In a normal pipeline, the buffers for the stages are all first–in–first–out (FIFO), and a stage can only execute one request at a time. This preserves the order of requests through the entire pipe. In a modern processor, the instruction pipeline must preserve this ordering. In a software pipe, we can allow stages to be replicated. However, if the order of the requests must be preserved at each stage, some additional mechanism must be used. Enforcing this ordering can be another factor that limits performance. We will examine this issue later, and show that removing ordering where it is not needed can improve the performance of the pipeline.

Evaluating the Work–pile–based Pipeline

The benefits of the Work–pile–based pipeline are the result of separating the threads from the stage objects. In other pipeline implementations, each stage object has a thread assigned to it that can only process work for that stage. If there is no work for the stage then the thread sits idle. This forced mapping of threads to stages has two side effects on the pipeline. First, any imbalance in the amount of processing in the stages causes threads to be idle while they wait for work from earlier stages. Second, load imbalances can only be addressed by replicating expensive stages to improve the throughput of the stage. This solution is ineffective in cases where individual stages are only slightly imbalanced relative to one another, but the overall pipeline suffers. Also, it cannot address cases where the source of the imbalance changes over time.

Separating the threads from the stages provides improved load balancing and scalability. The threads in the Work–pile–based pipeline search the request queues for outstanding work (the details of this search are discussed later in this section). This is analogous to a work pile with multiple work queues. Over time, the buffers for expensive stages will tend to have more work in them while less expensive stages will tend to have less. As a result, more threads will naturally find and process work for the expensive stages, and thus resources will be naturally dedicated to the expensive stages. Should the expensive stage change over time, as it does during ramp–up and ramp–down for the pipe, the threads will adjust accordingly. The result is improved load balancing. Scalability is improved because the execution model of the Work–pile–based pipeline is a pool of threads searching a set of queues for outstanding work to be processed. The number of threads in this pool is not dependent on the number of queues or number of stages. If there are relatively few processors, the user can select fewer threads than stages. If there are many processors, additional threads can be added into the thread pool. These threads will dedicate themselves to expensive stages automatically because of the balancing properties of the work pile execution model.

These benefits do have a cost to them, though. The first cost is that each stage in this new pipeline in an object, which must be instantiated for each request that arrives at the stage. Object instantiation can be slow, which can

```
while (not finished) {
    Find the next work item w from one of the buffers.
    nextStage = w.transform() ;
    while (no buffer for output of nextStage) {
        nextStage = nextStage.transform() ;
    } /* while */
    Place nextStage into the appropriate buffer.
} /* while */
```

Figure 4.22: Pseudocode for the actions of threads in the modified Work–pile–based pipeline.

be exacerbated if the heap lock must be obtained first.[2] Each work item must also be removed from and placed into a thread–safe buffer in the correct order, which also adds to the processing overhead. The next section discusses how this overhead can be reduced.

Refinements of the Work–pile–based Pipeline

The basic Work–pile–based pipeline can be refined by carefully noting the necessary ordering of work through the pipeline and only introducing ordering when it is necessary. This ordering can be implemented by adding ordering tags to each request and ensuring that a stage request will only be executed after all preceding ones have finished.[3] More importantly, it is possible to remove buffers between stages that have no ordering requirements, instead allowing the current thread to continue executing subsequent stages for the request. This reduces the cost of buffering work items as they flow through the pipeline.

This refinement is possible because the threads are separate from the stage objects. A thread in the basic Work–pile–based pipeline takes a work item from a buffer, executes the transform, and places the result into the next buffer. The key observation is that if there is no ordering needed between the requests for the result stage and its next stage, then the current thread can simply continue to execute the next stage as well. Buffers need only be introduced when ordering is required. To demonstrate the difference, the pseudocode for the worker threads in this modified Work–pile–based pipeline is shown in Figure 4.22.

If there is a long or expensive section of the pipeline that has no ordering, the modified pipeline may create a smaller number of large work items. The work pile execution model does not work well under these conditions. To create

[2]Some Java virtual machines provide a thread local heap for allocating small objects, and allocations to it do not require the heap lock to be obtained [27].

[3]This requires that the ordering tags have no gaps in them.

a larger number of smaller grained work items, non–ordering buffers can be inserted between stages. These buffers create additional work items for the threads but do not introduce any ordering between them. These buffers serve only as an optimization for a pipeline, and are not needed for correctness.

Thread Scheduling in the Work–pile–based Pipeline

One issue that has not been addressed is how to select a work item from the buffers. The most obvious choices are to enumerate over the buffers either forwards (checking buffers from left to right in Figure 4.21) or backwards (right to left).

Clearly, this choice affects the behaviour of the pipeline. Checking the buffers forwards favours new requests over those that have had a small amount of processing. Checking backwards favours requests that are already in the pipeline. Other schedulers could even assign a priority to the buffers to favour some stages over others. The best scheduling of the worker threads may depend on the needs of the application.

The Pipeline in CO_2P_3S

The Work–pile–based Pipeline has not yet been implemented in CO_2P_3S. As a result, the final form of the pattern template and the generated framework are not available. We are currently evaluating implementations of this new pipeline formulation.

4.2.4 Two–Dimensional Mesh

The Two–Dimensional Mesh pattern template provides support for mesh computations on regular, rectangular two–dimensional data. Typical mesh computations iteratively compute values for each data element based in its current value and the values of elements in some neighbourhood around it. This computation is repeated until the values converge to a final answer. Chapter 3 and Appendix C.1 provide considerable detail on this pattern template, and Appendix A provides complete source code for the reaction–diffusion example.

4.3 Comparing CO_2P_3S to Other Research

In Section 2.1 we examined other parallel programming research, and evaluated it using a set of 13 ideal characteristics of pattern–based parallel programming systems. Section 2.3 looked at relevant background work in object–oriented frameworks. This section compares and contrasts the PDP process and the CO_2P_3S parallel programming system with this other research.

Section 4.3.1 evaluates CO_2P_3S using the same 13 characteristics from Section 2.1. Section 4.3.2 relates the research in object–oriented frameworks to different aspects of our work in CO_2P_3S and the PDP process. Finally, Section 4.3.3 draws parallels between our work and research in object–oriented modeling languages.

4.3.1 Evaluating CO_2P_3S

One of the goals of this research was to build a tool that meets as many of the ideals of pattern–based parallel programming systems as possible. We believe that tools meeting these characteristics will have a better chance of being adopted by the programmer community.

This section evaluates CO_2P_3S using the same characteristics that we used to evaluate other parallel programming systems research in Section 2.1. We show how either the tool or the PDP process that it implements addresses these concerns.

Structuring the Parallelism

1. *Separation*: The PDP process preserves separation between the parallel structure and the application code by using frameworks. The generated frameworks encapsulate all of the application–independent parallel code into a set of abstract classes that invoke the application–specific code entered into the concrete classes by the user. The implementation of the parallel structure can evolve independently of the user application.

   Ultimately, the application code is tied to the parallel structure through the interface for the hook methods. For instance, the interface for the hook methods in the Distributor framework is different from the interface for the hook methods in the Mesh framework. These interfaces are tied to the pattern semantics, which in turn must match the application semantics in order to successfully create a parallel program with the template. Unfortunately, there may be a mismatch between the interface for the hook methods in the framework and existing sequential code that implements the problem. This is most easily remedied by using an Adapter pattern [37] to translate method invocations on objects in the generated framework to method invocations on objects from existing sequential code. This was the approach taken in the reaction–diffusion example program. This is also one of the suggestions for addressing the problem of composition with legacy code in Section 2.3.2.

2. *Hierarchy*: The frameworks in the PDP process rely on object composition to hierarchically create larger parallel structures. The hook methods for any given framework can instantiate other frameworks and use these as normal collaborating objects through delegation, just as in normal sequential programming.

   This method of composition works because the generated frameworks do not monopolize the flow of control over the whole application, but rather manipulate data using a specific parallel structure and return the results. This style of interaction avoids some framework composition problems. The remaining composition problems are discussed in Section 4.3.2.

3. *Independence*: The frameworks generated for CO_2P_3S help ensure independence by encapsulating all of the details of the parallelism behind a well–defined interface. Encapsulating these details prevents a framework from relying on the internal details of another, making them independent of one another.

4. *Extendible*: There is nothing inherent in the PDP process to prevent users from adding their own patterns to a system. This is a function of the system itself. CO_2P_3S itself is implemented as a framework to allow new pattern templates to be added to the system. To simplify the process, CO_2P_3S will include tool support for creating new pattern templates [18]. This work will improve upon the extendibility provided in DPnDP by allowing pattern–specific behaviour to be added to the abstract framework classes.

   In contrast, the patterns supported by other pattern–based systems are usually fixed by the tool. If the problem cannot be efficiently solved with some combination of the supported patterns, the programmer is forced to find a different tool. DPnDP provides some support for extending its set of patterns, but the functionality of the new patterns is limited compared to the patterns supplied with the tool.

5. *Utility*: The initial set of patterns supported by a tool will dictate the range of problems that can be solved. This initial set must cover a broad range of problems even if the tool allows new patterns to be added. New users will expect to be able to use the tool to solve their problems immediately; only advanced users will take the time to add new pattern templates.

   We continue to improve the utility of CO_2P_3S by mining parallel design patterns from new applications. Design pattern templates can be created from these mined patterns and incorporated into CO_2P_3S. Once tool support is in place and users are able to create (and possibly even exchange) their own patterns, the bottleneck will be in identifying the patterns, not implementing them.

   We can also use new applications to evolve our existing pattern templates and frameworks. These applications may require alternative structures that cannot be expressed with the set of parameters, or may require additional hook methods that are not provided by the framework code. We may wish to modify the templates and frameworks to accommodate these needs.

6. *Openness*: The layered programming model used by the PDP process supports our augmented definition of openness, where programmers should be able to access the complete system to modify and improve their programs. The Patterns Layer concentrates on the creation of correct parallel programs by encapsulating many of the details of the parallel structure. This approach generally sacrifices performance, though. The structural code is generic as it must be applicable to a broad range of applications. The Intermediate Code Layer provides access to the structure of an application in a high–level parallel programming language, which can be used to both tune the application or create structures that cannot be expressed using the pattern templates and their parameters. Finally, the Native Code Layer provides access to all code supporting the abstractions at the layers above it.

   The primary benefit of the layers is the structured manner in which they expose the details of the system to the programmer. Each layer hides the details of the one underneath. Thus, programmers can select an appropriate layer based on the problem they are experiencing and work at that layer without becoming overwhelmed with irrelevant details. Thus, the layered approach should benefit both novice and experienced users.

   In contrast, the closed implementations and closed programming models of other systems prevents programmers from making the kinds of changes that are permitted in our open model. The implementation of a given parallel structure cannot be improved or tailored for the specific program, which limits performance and restricts the set of problems that can be implemented.

Programming

7. *Correctness*: The Patterns Layer in the PDP process addresses correctness using a combination of code generation and encapsulation.

   Code generation is used to create correct framework code for the pattern templates selected by the user. The resulting program is guaranteed to match the selected parallel structure. Also, the framework code correctly implements all of the parallel code, including concurrency, communication, and synchronization. This saves the user from having to write and debug this difficult and error–prone code. Further, the hook methods for the framework can be implemented as normal sequential code. The structural code does not rely on the user to correctly implement any parallel code for it to work correctly. The user can concentrate on their problem rather that the parallel structural code that will execute it.

   Encapsulation hides both the parallel structure and the implementation details of this structure from the user. The abstract classes in the generated framework code contain all of the parallel structural code, which cannot be accessed at the Patterns Layer. This prevents the programmer from accidently changing this code and introducing errors. Further, the framework instantiates all of the necessary objects that it needs rather than relying on the programmer to do this. The user does not need to be concerned with the implementation details of the parallel structure unless the lower layers are used.

The CO_2P_3S user interface also addresses correctness concerns beyond supporting the basic characteristics of the Patterns Layer. The user interface ensures that all template parameter values are valid before the template is used to generate code. In particular, the interface ensures that all necessary parameters are specified, all class names are unique, and all other parameters have reasonable default values. This way, the generated frameworks will always correctly implement a given structure. The user interface can also help enforce encapsulation. Code viewers (Section 4.1.4) are offered on only those classes that contain any hook methods or might otherwise need to have methods added to them. The remaining classes (the abstract structural classes and possibly other helper classes) cannot be modified through the interface. The CO_2P_3S interface can also prevent other, more general programming errors. For instance, the combination of the viewers and code editor dialogs (Section 4.1.4) provide stubs for the hook methods, so the user does not need to determine the correct methods and their signatures. These dialogs also provide no opportunity for the user to edit the signatures and introduce any incompatibility between the application–independent structure and the application–specific classes implementing hook methods.

8. *Language*: Before we evaluate the use of an existing, familiar programming language, note that the PDP process is independent of the programming language and underlying parallel architecture. The frameworks and tool interface may change to accommodate other concerns in different languages and architectures, but the process remains the same.

   We purposefully stray from the ideal of preserving the syntax and semantics of the original programming language as this can limit the potential concurrency in a program [119]. In languages that support run–time exceptions, for example, statements in a program must generally execute in order so that an exception handler can correctly determine the state of the program. Some reordering of instructions is possible, but an instruction that can potentially throw an exception cannot be moved past another such instruction. If it is, then the optimized program may throw the wrong exception from the perspective of the programmer. Unfortunately, an analysis of Java programs suggests that a large percentage of instruction can potentially throw exceptions, limiting code motion [43]. For example, in Figure 4.23, the two statements in the `try` block cannot be reordered without changing the semantics of the program. As

```java
int[] array = new int[5] ;
Object obj = null ;

try {
    // Should throw ArrayIndexOutOfBoundsException
    array[5] = 1 ;
    // Should throw NullPointerException
    obj.toString() ;
} catch (ArrayIndexOutOfBoundsException aioobe) {
    . . .
} catch (NullPointerException npe) {
    . . .
}
```

Figure 4.23: A Java code example where statement reordering can change the semantics of the program.

written, the code should throw an `ArrayIndexOutOfBoundsException`. If the statements are reordered, a `NullPointerException` is thrown instead. However, the program should never reach the second statement, so this is the wrong exception.

A concurrent program can execute statements that would not have been executed in the sequential program, resulting in similar problems. However, this concurrency is necessary for high performance parallel programs, so changes to the semantics of the programming language are unfortunate but necessary.

There has been some compiler research aimed at removing this ordering restriction for instructions that throw exceptions at the cost of more expensive exception handling when errors occur [43]. The compiler generates two versions of the code. The *optimized code* ignores dependencies caused by exceptions, which enables other optimizations to be applied more aggressively than would normally be possible. This handles the common case where no exceptions occur. However, if an exception is thrown, the optimized code may throw the wrong one (as would happen in Figure 4.23) and the program state may be different than the original code. To correct this, the compiler also generates *compensation code* as an exception handler for the optimized code. This compensation code fixes the program state and throws the exception that would have been thrown by the original code. The compensation code is generated based on an analysis of the live variables in the exception handler body, and takes advantage of the fact that most exception handlers display simple error messages and stack information, and never access program state.

It can be argued that we will not violate this characteristic in programming languages that directly support threads and have existing semantics for concurrency, such as Java, unless a system requires additional semantics to support its programming model. However, from the perspective of programmers who write sequential code, it can be argued that the semantics have changed to accommodate concurrency. Java programmers that do not use threads will not immediately see the need for synchronized methods, for example. This latter viewpoint cannot be ignored as it will be common in users who are new to parallel programming. Further, not all programming languages have semantics for concurrency. Thus, from the perspective of the overall process, we must consider the semantic differences between sequential and concurrent programming. Individual implementations of the PDP process may leverage existing concurrent semantics in the chosen base language if they are available.

9. *Non–Intrusiveness*: The use of frameworks prevents the PDP process from meeting this characteristic. Frameworks are intrusive by nature, imposing a design structure that is an integral part of the framework and hence is difficult to change. However, in exchange, frameworks provide the possibility of large–scale reuse, reusing the design over many different programs. This design determines what objects are part of the program, how they collaborate, the set of hook methods that the framework makes available, and the location of these hook methods in the overall flow of control through the structural code.

User Satisfaction

10. *Performance*: The layered programming model helps address the performance characteristic. The generated frameworks are correct but conservative in their implementation, so that they can be used across a broad range of applications. The lower layers in the PDP process provide openness, allowing access to the framework structure and the class libraries used to support this structure so knowledgeable users can locate and remove any performance bottlenecks. In contrast, other systems are closed, not allowing users to tune any system–specific code.

    While the frameworks generated at the Patterns Layer must be conservatively correct, they must still provide some performance benefits. Without performance gains at the highest layers of a tool, there is little incentive for users to expend effort to learn to use it. We cannot expect users, particularly novice users, to be required to use lower layers of the PDP process to produce parallel speedups for applications. However, we can expect the performance gains to be commensurate with the amount of effort that the user puts in.

    To show that CO_2P_3S templates can provide speedups at the Pattern Layer, even without tuning, we present the performance results from the

Table 4.1: Speedups and wall clock times for the reaction–diffusion example program.

	Processors	2	4	8	16
1680 by 1680 surface	Speedup	1.75	3.13	4.92	6.50
	Time (sec)	5374	3008	1910	1448
	Std. dev.	229	96	105	97

reaction–diffusion example from Chapter 3 in Table 4.1. These results were gathered using a native–threaded Java implementation from SGI (Java Development 3.1.1, using Java 1.1.6) with both optimizations and JIT turned on. The program was executed on on SGI Origin 2000 with 44 195MHz R10000 processors and 10GB of memory. The virtual machine (VM) was started with 512MB of heap space. The speedup numbers are based on wall clock times, compared to a sequential implementation of the same problem executed using a green–threaded VM. We initially believed that using a native–threaded VM might have extra overhead that would cause the sequential program to have longer execution times and skew the results. However, the performance of both VMs on a single thread was identical. The times for the green–threaded VM was used because there is more performance data for that version.

Note that the times in Table 4.1 include only the computation time; initialization and output are not included. The problem scales up to about four processors, but performance starts falling off as more processors are added. The problem is granularity; as more processors are added, the amount of work assigned to each falls until synchronization costs begin to limit the overall performance. Larger computations, with a larger surface size or with a more complex computation for each mesh element, will yield better speedups for larger numbers of processors. Using an unordered mesh would remove some of the synchronization costs at the expense of non–determinism in the results and possible difficulties in evaluating the termination conditions.

11. *Support*: Since CO_2P_3S is a relatively new system, it does not have a complete set of support tools. It currently has tools for compiling and running programs, but lacks support for debugging and performance evaluation. These tools will be the subject of future research.

12. *Usability*: The speedup numbers for the reaction–diffusion example are not necessarily the best that can be achieved. It is tempting to be disappointed with the results. However, they do show that the Mesh pattern template can produce performance benefits even without any tuning.

    Another important consideration, also generally ignored by the parallel programming systems community, is the development costs of writing parallel programs. In particular, the time needed to develop an application is rarely considered by tool developers [111]. Outside of the parallel programming systems community, programmers are interested in the answers generated by their programs, not the speedup obtained by the parallel version. These users will not spend the time to write a parallel program if the development costs outweigh the performance benefits.

For our example, the complete, correct parallel structure was generated within minutes using the Mesh pattern template. Using classes from an existing sequential version of the program, the complete application was finished in several hours. For this limited effort, the performance results are encouraging.

We have gathered additional data on the usability of CO_2P_3S from a user study. The experiment and the results are discussed in detail in Chapter 6. The results suggest that users write significantly less application code using CO_2P_3S compared to writing a Java solution, mainly because the parallel structure is generated. Also, the application code written by CO_2P_3S users is less complex than that written for an equivalent Java program.

The results of the usability study reinforce our experiences with the reaction–diffusion example. The complete program, including the generated code, was 489 lines of Java code[4], of which 236 (48%) is user application code. Over half of the code is dedicated to the parallel structure. A correct implementation of this code is generated for a CO_2P_3S user. Of the 236 lines of application code, 183 lines (78%) was reused directly from the existing sequential version. We must point out that these numbers are a function of the problem being solved, and are not an inherent property of all programs written using CO_2P_3S.

13. *Portability*: There are two ways that this research can address the portability problem.

    First, different versions of the Native Code Layer can be supplied for different parallel architectures. The benefit to this solution is that the programmer can change architectures by simply linking in a different version of the Native Code Library. There are several issues with this approach. It will require that the interface to the Native Code Layer be fixed so that the different implementations are easily interchangeable. Also, the framework code must be written independently of the architecture. For example, all data exchanges will need to be done through explicit communication primitives, even in shared–memory machines where this can

[4]Lines of code is determined by counting semicolons in source code that is stripped of comments. This does not correct for `for` statements, which count as two lines of code.

be done through shared variables. Finally, this solution limits the possibility of applying different optimizations for different architectures. Only optimizations that apply to all architectures can be safely applied to the frameworks at the Patterns Layer.

The second possibility is to generate different framework code for each architecture. The target architecture can be an extra parameter for each pattern template or a setting in the tool. The primary benefit to this approach is that the framework code can be better optimized for its execution environment. This is similar to the use of architecture–specific information in P^3L (Section 2.1.1, the *portability* characteristic). A drawback of this approach is that the framework code must be regenerated for each architecture. This regeneration will overwrite any changes made to the structural code made at either the Intermediate Code or Native Code Layers. If a program must execute on multiple architectures, the user may keep different versions of the program, which can lead to maintenance problems.

This section demonstrated how CO_2P_3S, implementing the PDP process, satisfies more of the 13 characteristics of ideal pattern–based parallel programming systems than existing systems. In particular, this work advances the state of parallel programming systems research with regards to the *openness* and *correctness* concerns. Also, even though CO_2P_3S is still in the early stages of development, we are considering its *usability* in the hands of real users so that it can be improved.

Even our approach is incomplete with respect to these criteria though. The *language* and *non–intrusiveness* characteristics are not satisfied, and the *utility* of the system will be under constant improvement.

4.3.2 Frameworks and the PDP Process

This section relates the PDP process to the research in object–oriented frameworks from Section 2.3. Note that the framework research can be related to aspects of the PDP process other than the frameworks generated at the Patterns Layer. The rest of this section explores this relationship in more detail.

Documenting Frameworks

In Section 2.3.1, hooks were discussed as a means of documenting how to use a framework to accomplish a specific task in an application program. One of the characteristics of a hook is the level of support that the framework has for it. To recap, the levels of support, from most support to least, are:

1. Option hooks. The user can supply one of a number of pre–built components for some aspect of the framework.

2. Supported pattern hooks. The framework defines an interface for fulfilling some requirement, but the implementation of that interface is application–specific and must be provided by the user.

3. Open–ended hooks. These hooks show the user how to make changes to the framework that are not supported by either of the other two types of hooks. These changes may involve structural changes to the framework classes.

These levels of support can be related to different parts of the PDP process spanning the three layers. The relationship is:

1. Option hooks. The hooks correspond to the pattern template parameters in the Patterns Layer. The values of these parameters are used to generate framework code for that specific version of the template. The framework generator may generate a different implementation of a component in the structural code to accomplish this, such as generating different Strategy objects for different topologies in the Mesh framework (Section 3.2.4). Alternately, the framework generator can conditionally include different method bodies and signatures for different parts of the framework. The result is that the template parameters alter the structure of the generated framework in an analogous manner to supplying a different pre–built component when instantiating a framework.

   For tools supporting the PDP process, the user needs to know nothing about the internal details of the framework to use these hooks. The changes to the frameworks are handled by conditional code generation and are generally limited to classes that are not accessible to the user at the Patterns Layer. This is not true of all frameworks, though. There may be other effects of parameter changes that appear in the hook methods in a given framework. For example, in the Mesh framework the set of operation methods and their signatures are determined by a combination of two template parameters, the topology and the number of neighbours.

2. Supported pattern hooks. The supported pattern hooks correspond to the hook methods in the generated frameworks. The structural part of the framework code defines the expected interface, which the user implements in an application–specific way. In CO_2P_3S, a class with stub methods for this interface is generated. This class is displayed in the code viewers from Section 4.1.4.

   The use of these hooks requires more detailed knowledge about the internal details of the framework. For the Mesh framework, it is important to understand how the hook methods are invoked in the overall flow of control through the mesh computation. A user will not be able to write applications using the framework without this information. However, the user does not need all of the details regarding how the methods are invoked; the relative order is sufficient. When implementing the operation methods for the Mesh, the user need only know when they are called relative to the other hook methods. It is not necessary for the user to understand the use of the Strategy pattern, which helps determine the correct method to execute for a given mesh element, to write this code.

3. Open–ended hooks. As with other frameworks, there is no specific support for open–ended hooks in CO_2P_3S frameworks. These hooks can only be introduced through additional documentation describing the implementation of the frameworks, which will need to be supplied with CO_2P_3S. Unlike other parallel programming systems research, the Intermediate Code and Native Code Layers provide access to the structural framework code, which permits the user to make changes for these hooks.

This relationship reinforces an aspect of tool support for programming with frameworks that we have already discussed, which is providing the means to use option hooks without any knowledge of the framework structure. In normal framework programming, option hooks still require some knowledge of the structure so that the appropriate objects can be created and assembled into a complete program. The template parameters, which correspond to the option hooks, can be used to guide a code generator to address this problem automatically. As well, tool support can help ensure correctness; the tool can ensure that the user selects one option out of the set of legal ones, which reduces the probability of error.

We can also apply tool support to reduce the probability of user error for supported pattern hooks. Generating concrete subclasses for the framework and stubs for the hook methods saves the user from having to derive this information about the framework. More importantly, it saves the user from having to enter this code. For the Mesh frameworks, it may be difficult to determine both the set of operation methods that are needed and their signatures. The signatures present an additional problem in that the order of the parameters may be important. For instance, a user could write

```
void interiorNode(MorphogenPair up,MorphogenPair down,
        MorphogenPair left,MorphogenPair right) ;
```

as the signature of the `interiorNode()` hook method. The parameters in this signature are in the wrong order. The correct order, from Figure 3.6, is `right`, `left`, `up`, and `down`. The compiler will not be able to catch this error as the argument types match. If the computation depends on the location of the neighbour (for example, in an *anisotropic* reaction–diffusion example, where the morphogens diffuse at an angle rather than horizontally and vertically), the results will be incorrect. We further reduce programmer errors by using template viewers and code editors, so the user cannot accidently modify these signatures.

Composing Frameworks

Although the work in this dissertation does not specifically address framework composition, we must consider the issues involved. Even small parallel programs may require multiple parallel design patterns to create an efficient implementation, and this will require the composition of multiple frameworks. This section examines properties of the frameworks created by CO_2P_3S that can alleviate composition problems. Some of these properties are a function of the PDP process while others are specific to CO_2P_3S.

The five primary compositions problems, from Section 2.3.2, are:

1. composition of framework control,
2. composition with legacy systems,
3. framework gap,
4. composition overlap of framework entities, and
5. composition of entity functionality.

Composition of framework control is addressed by the nature of the frameworks in CO_2P_3S. Parallel programs, just like their sequential counterparts, accept input, compute results, and create output (normally the computed results). The primary difference is the use of multiple processors to compute the results. In the end, though, the results must be made available. As a result of this style of computing, no single framework will monopolize the flow of control for the complete lifetime of the program. Thus, composing frameworks is done by allowing the hook methods in one framework to instantiate and use another.

Composition with legacy systems is best addressed using adapters, just as originally discussed in Section 2.3.2. The reaction–diffusion example took this approach, using the generated `MorphogenPair` class as an adapter for classes from the sequential version of the program.

In the context of the PDP process, framework gap would refer to the inability to create the desired parallelism for some portion of an application. This can be addressed at lower layers by inserting the extra parallelism. More generally, though, this indicates either a missing design pattern or a gap in the pattern template parameters that prevents the user from selecting a useful member of the pattern family. Ongoing research in allowing users to add patterns, which also includes the ability to modify existing ones, may also address framework gap.

Composition overlap of framework entities is primarily the result of the lack of a standard definition of the domain of a framework. Most frameworks define semantics for real–world entities in the application, which leads to overlap when different frameworks define conflicting semantics for the same entity. The frameworks for CO_2P_3S define the concurrency in a manner that is independent of the semantics of the entities in the final application. The Mesh framework, for instance, defines the concurrency for a mesh computation but does not define the semantics of the data on which the computation is applied. If the other frameworks in CO_2P_3S also exhibit this characteristic, the possibility of overlap is reduced.

Finally, the composition of entity functionality is not addressed. If necessary, the user can apply the existing solutions at a lower layer.

Instantiating Frameworks

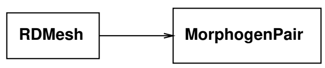
Section 2.3.3 discussed CORRELATE, a concurrent object–oriented language. The interesting feature of this language was the use of language support to create necessary classes and instantiate the Active Object framework used to provide the concurrency. The concern was that the large number of classes used in the framework would be difficult for a user to manage and would require knowledge of the framework structure. To address this problem, extra syntax was introduced to indicate both the active objects and any method invocations on these objects. This syntax was used by a preprocessor to create the complete framework for each active object and transform all method invocations on these objects. In addition, CORRELATE provides a metaobject protocol to expose other important abstractions in the Active Object framework to the user.

CO_2P_3S has similar concerns with its frameworks, particularly at the Patterns Layer. The premise behind this layer is that the user should be able to concentrate on their problem and not on the parallelism that will be used to execute the final program. This idea extends to the process of instantiating the framework. The internals of the frameworks should be encapsulated until they are needed at the lower layers.

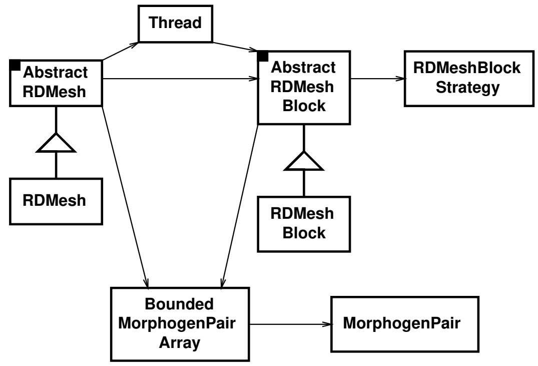
The difference is in our approach to the problem. Rather than using extra syntax and a metaobject protocol, we use a combination of encapsulation and conditional code generation to solve this problem. Our generated frameworks use encapsulation to limit the user's view of the internal structure, and generate code to instantiate all of the needed objects without requiring any user code. Like CORRELATE, we still need to expose some of the abstractions in the encapsulated code. Rather than a metaobject protocol, we again use conditional code generation to specialize the framework based on the values of the pattern template parameters.

4.3.3 Object–Oriented Modeling Languages and the PDP Process

This section relates CO_2P_3S and the PDP process to the research in object–oriented modeling languages discussed in Section 2.4.



(a) The user view of the Mesh framework at the Patterns Layer.



(b) The user view of the Mesh framework at the Intermediate Code Layer.

Figure 4.24: The user views of the Mesh framework, at both the Patterns Layer and the Intermediate Code Layer.

CO_2P_3S and UML Tools

The primary difference between CO_2P_3S and the UML modeling tools is that CO_2P_3S models are patterns and not the more general models supported by UML. Patterns have much more context and structure than general models. Through encapsulation, we ensure this structure matches the chosen pattern during the initial stages of program development. Lower layers open up the details of the framework for the user. A UML model does not have the same contextual information.

Another way to view the distinction is to consider the user model of the frameworks at different development layers, particularly the Patterns Layer and the Intermediate Code Layer. At the Patterns Layer, the model of the framework for a pattern template consists of only a few classes, such as the Mesh example in Figure 4.24(a). This model is specialized using pattern template parameters rather than by modifying the classes in the model directly. The programmer is limited to adding hook method bodies to the `Mesh Element` class and instantiating instances of the `Mesh` class. The Intermediate Code Layer expands this model to include more detail, shown in Figure 4.24(b). At this lower level, the generated code can be considered as a UML model that a programmer can change and manipulate directly.

Unlike the work in generating code from UML given in Section 2.4.1, we are generating code from a pattern specification. This pattern specification describes a strategy for parallelizing an application. It does not describe the implementation of that pattern. At the Patterns Layer, the important characteristic of a pattern template is its behaviour, not its implementation. The implementation of the pattern is not important until the user enters a lower layer that exposes it.

Since we are generating code from a pattern, rather than a general UML model, we do not need to support the full range of design elements found in UML. This simplifies our code generation task. However, it is possible to take advantage of the ability to generate code from UML designs by creating a UML model from a pattern template (and its parameter values) and then generating code from the model. It may be possible to use the same UML model to generate code in different programming languages. However, translating the design elements to different languages may incur overhead based on the available language features. Further, this translation may result in code that does not conform to generally accepted programming practices for a given language. For example, the Java code generated for a UML model that uses multiple inheritance will likely appear awkward to a Java programmer, who would not have designed the program that way in the first place. Since the target of our generated code is the programmer, this is undesirable. It is imperative that the programmer be able to understand and modify the generated code at lower layers.

We have chosen the code generation approach over the executable model approach in this work. Parallel programming is a performance–driven field, and the overhead of executable models is unacceptable for final, production code. However, the executable model approach could still be used to quickly explore different pattern templates for a problem during initial design stages.

Recall that the UML virtual machine used four layers to describe its architecture, where each layer describes the previous layer. The PDP process uses layers in the more traditional sense, where each layer builds up higher–level services or abstractions using the previous layer. However, it is possible to consider the layers as a description of the generated frameworks as described earlier. Each lower layer provides a more accurate description of the details of the generated code. The difference is that the highest layer of the UML virtual machine describes its modeling language, where our layers only describe the individual frameworks.

The PDP Process and the ROOM Modeling Language

The conceptual framework of ROOM is the combination of two paradigms that partition the modeling and implementation space of a real–time, object–oriented application. Parts of this framework have parallels in the PDP process, but there are also some important differences.

One similarity at the abstraction levels paradigm is the use of separate abstractions for different aspects of a system. Within ROOM, these abstractions separate concurrency considerations from the development of data structures. Within the PDP process, each layer provides abstractions specifically targeted at different stages of program development.

The modeling dimensions paradigm of ROOM can also be applied to the PDP process. Each of the modeling dimensions (structure, behaviour, and inheritance) are applied to the abstractions level in a different way. Similarly, these three dimensions are applied to the PDP layers differently. At the Patterns Layer, these dimensions are specified by the pattern template parameters, which control code generation. Behaviour is further specified by the hook methods provided by the frameworks. At lower levels, as the implementation of the frameworks is opened, these dimensions come under the control of the programmer.

The navigational aspects of the conceptual framework of ROOM have analogies to the layers of the PDP process. The layers provide the programmer with a path from a high–level view of the patterns (at the Patterns Layer) to successively lower–level views (the Intermediate Code and Native Code layers). Following this path eases the transition to the low–level run–time support code for programmers.

An important difference between the ROOM modeling language and the PDP process is in how the different abstractions are used during program development. The paradigms (and associated layers) in ROOM partition the modeling space. The designer can incrementally work through the modeling space during the creation of a system, but must address all of the concerns for each paradigm to complete the system. In contrast, the programmer decides which layers of the PDP process are applicable based on the requirements of the program. For example, the programmer only needs to work at the Intermediate Code Layer if the Patterns Layer Code does not meet performance requirements or does not support a needed pattern structure.

Another difference is that the concerns for each paradigm in ROOM are interrelated. Each describe the same system, and as such must be consistent with each other. In the PDP process, the lower layers expose details that are initially consistent with the higher–level view. However, the programmer is able to change the implementation at lower layers. As a result, while the lower–level code should still implement the same pattern, the code is no longer constrained by the pattern template parameters and thus may not be consistent with code that can be generated at the Patterns Layer.

4.4 Summary

This chapter examined the CO_2P_3S parallel programming system in more detail. CO_2P_3S is our concrete implementation of the PDP process. An important aspect of the PDP process is that it does not dictate every aspect of any system that implements it. Each system must make some choices regarding how it will support different aspects of the process. CO_2P_3S demonstrates some of our choices, but different systems will make different choices.

This chapter started by highlighting the graphical user interface of CO_2P_3S, showing some of the ways a tool can address correctness issues beyond those specifically addressed in the PDP process. The design patterns supported by CO_2P_3S were discussed next. Finally, CO_2P_3S was compared to the research from Chapter 2. The system was evaluated using the 13 characteristics of an ideal design–pattern–based system, where we showed that it addresses more of these concerns than existing systems. We also examined our tool with respect to the work in object–oriented frameworks and modeling languages, and found relationships between that work and different aspects of both CO_2P_3S and the PDP process.

Chapter 5

Example Applications in CO_2P_3S

One of the characteristics of a good pattern–based parallel programming system is its *utility*, which means that the patterns supplied with a system should cover a broad range of applications. In systems that are intended for writing general–purpose parallel applications, such as CO_2P_3S, this characteristic has the added caveat that the patterns cannot be limited to a particular application domain. In essence, the patterns supported by a system must exhibit the *utility* characteristic themselves.

The previous two chapters covered the PDP process and the CO_2P_3S parallel programming system. To this point in this dissertation, both the development process and our implementation of that process have been illustrated with a single application, the reaction–diffusion texture generator written using the Mesh design pattern template. This chapter discusses three additional applications developed using CO_2P_3S that show the other pattern templates and illustrate the composition of the generated frameworks. Each application was written using the facilities available at the Patterns Layer. These applications were run on several different shared memory systems to ensure that CO_2P_3S works on a variety of architectures.

Section 5.1 presents an implementation of the Parallel Sorting by Regular Sampling algorithm [94]. This example serves two purposes. First, it is an explicitly parallel sorting algorithm with no sequential equivalent. This demonstrates that CO_2P_3S is not limited to parallelizing existing sequential code. Second, it demonstrates the composition of CO_2P_3S frameworks, as this application uses two instances of both the Distributor and Phases templates.

Section 5.2 details a program that solves instances of the 15–puzzle using parallel iterative–deepening A* search [58]. The program uses the Distributor template to parallelize the search for a given depth, together with a dynamic frontier to help with load balancing.

Section 5.3 discusses an implementation of parallel JPEG compression [51]. This example uses the Work–pile–based pipeline from Section 4.2.3 to convert GIF files into JPEG files. This new pipeline is also compared to the traditional pipeline (Figure 4.18, page 87) to show that the Work–pile–based model improves performance.

5.1 Parallel Sorting by Regular Sampling

5.1.1 Problem Description

Parallel Sorting by Regular Sampling (PSRS) is a parallel sorting algorithm that provides good performance over a broad range of parallel architectures [94]. The primary strength of this algorithm lies in its load balancing strategy, which samples the data to try to find pivot values that evenly distribute the data across the processors.

An important aspect of PSRS is that is it an explicitly parallel algorithm that has no acceptable sequential counterpart. Parallel versions of good sequential sorting algorithms, such as Quicksort and Mergesort, have proven to be suboptimal solutions. For example, the performance of straightforward parallel versions of the Quicksort algorithm are limited by two problems. First, the initial partition of the complete data set is a large sequential operation that limits the speedup, as anticipated by Amdahl's Law [4]. Second, it takes time before all of the processors are involved in the computation. In naive implementations, only about half of the processors are actually sorting while the rest wait for results. This idle time also limits the speedup. Because of these problems, the speedup that can be achieved is limited to around 5 or 6 regardless of the number of processors used. In contrast, PSRS has no initial sequential stage, but rather divides work among all processors at the very beginning of the algorithm. The load balancing strategy reduces the idle time of the processors.

The PSRS algorithm consists of four phases, illustrated in Figure 5.1. Each phase must be completed before the next starts. The phases, executed on p processors, are:

1. In parallel, divide the input array into p contiguous lists and sort each list using Quicksort. Select $p-1$ evenly spaced sample elements from each sorted list.

2. Select a designated processor to sort the complete set of sample elements. Choose $p-1$ evenly spaced pivot elements from the sorted sample set.

3. In parallel, partition each sorted list into p sublists using the pivot values.

4. In parallel, merge the partitions and store the results back into the original array.

This implementation of PSRS differs slightly from the above description to take into account a shared memory execution environment in two ways. First, the data does not need to be physically partitioned but can instead be split

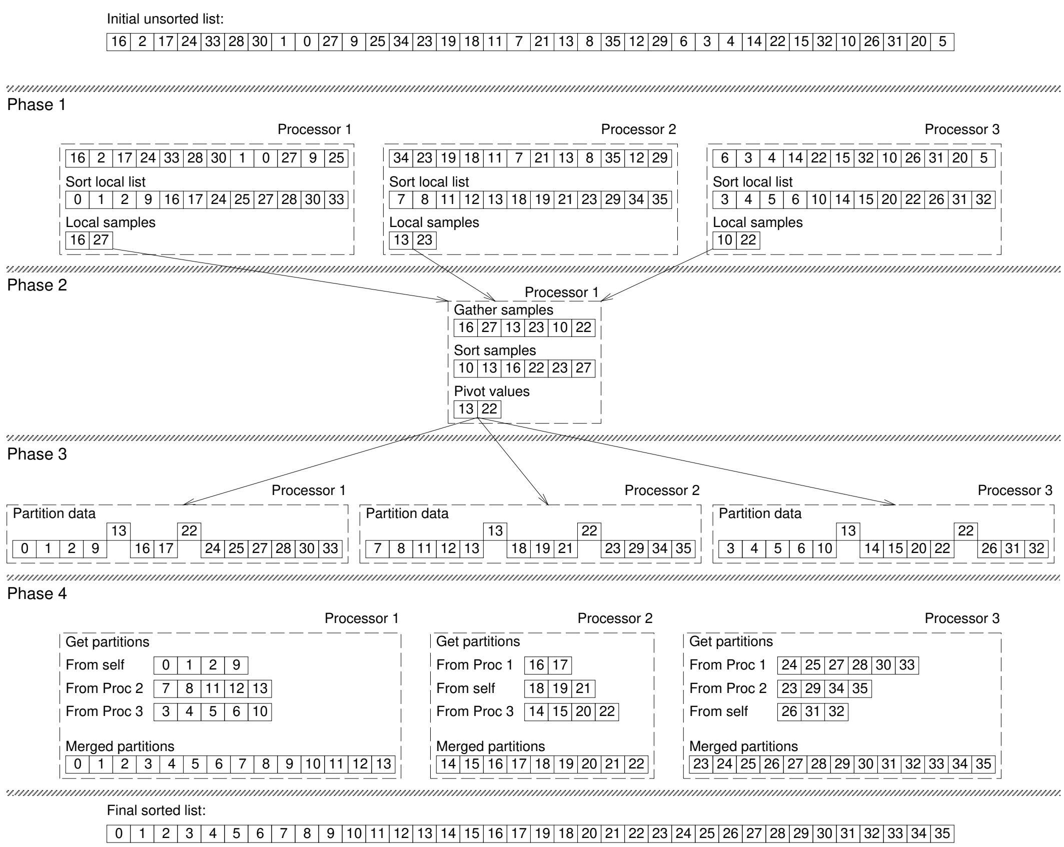


Figure 5.1: An example of Parallel Sorting by Regular Sampling.

using ranges, which are analogous to the bounded arrays used in the Mesh framework (Section 3.2.4). The range is defined by a pair of indices, *lower* and *upper*, which define a subarray on a much larger array. Each range shares a copy of the original data. Accesses to the data elements in the range are translated to data elements in the original array based on the lower range index. These ranges can be used to divide the input array without copying it.

Second, the third and fourth phases can be implemented differently as each processor has access to the complete data array. The last two phases are rewritten as:

3. In parallel, using the values of two consecutive pivot values, find the range in each sorted sublist that is bounded by the two pivots. Markers are prepended and appended to the pivots to indicate the first and last values. Note that, unlike the previous version of PSRS, this phase does not require interprocessor communication because the data is no longer physically partitioned across the processors. Instead, each processor has access to the complete data array, and can create the necessary ranges by scanning the lists created in the first phase.

4. In parallel, merge the data in the list of ranges created in the previous phase and store the results back into the original array. In a shared memory environment, this phase has a data dependency that must be considered. A processor cannot start writing data back to the original array until the merge is complete. Otherwise, the merging phase can read incorrect data. This phase needs to be rewritten as two subphases:

   4.1. In parallel, merge the ranges into a temporary buffer.

   4.2. In parallel, store the results in the original array by copying the temporary buffer.

The previous example, using the new formulation of PSRS, is illustrated in Figure 5.2.

5.1.2 Pattern Selection

The parallelism in this algorithm is clearly specified in the algorithm description from Section 5.1.1. PSRS consists of a set of phases, with some executing in parallel. For the parallel phases, a fixed number of processors execute the same operations on different portions of the data.

The implementation of this program uses two instances of the Distributor pattern template and two instances of the Phases pattern template. The first instance of the Phases template, `PSRSPhases`, creates the unsorted data, implements the four phases of PSRS, and verifies the results. The last phase of the PSRS algorithm in `PSRSPhases` creates and uses the second Phases template, `MergePhases`, to implement the two subphases. Alternately, the

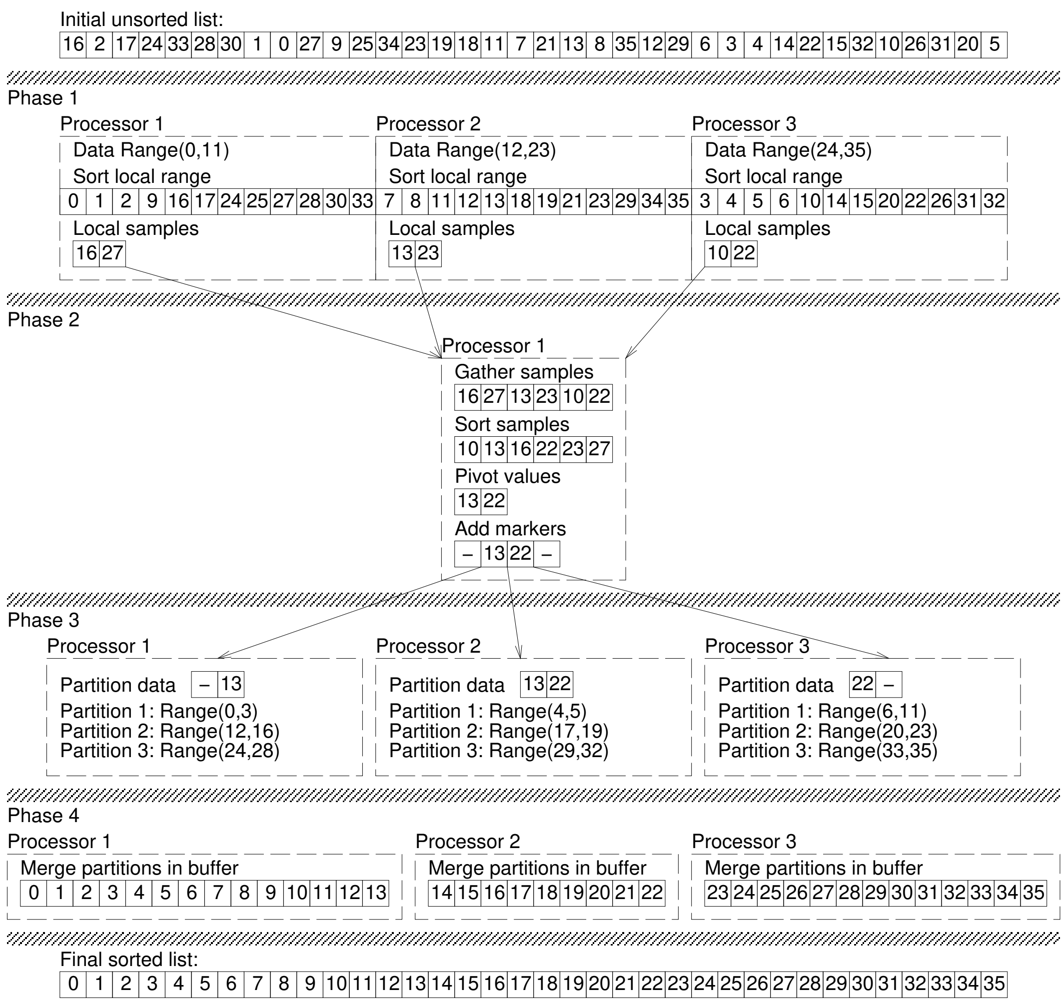


Figure 5.2: The shared memory version of PSRS, using ranges rather than physically distributing data across processors.

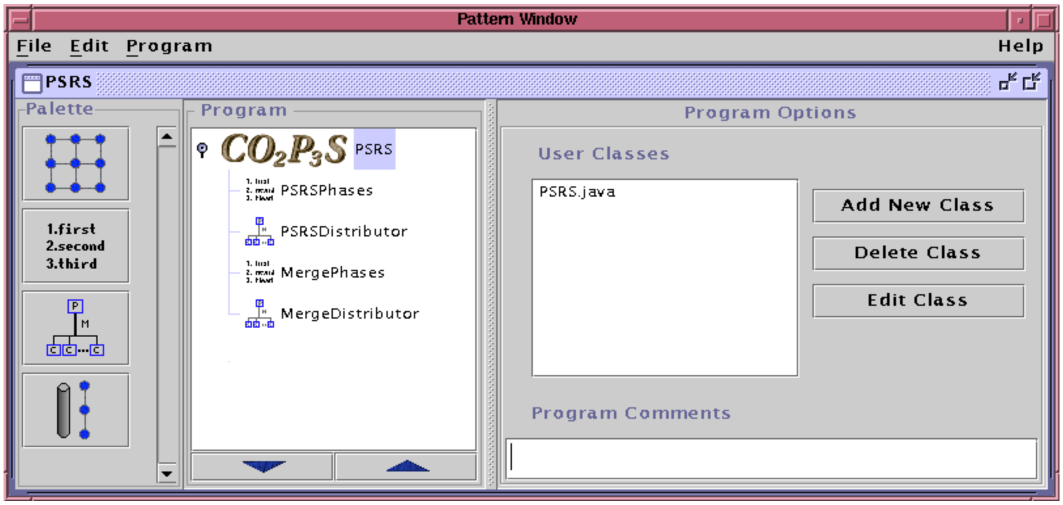


Figure 5.3: A screenshot of the CO_2P_3S implementation of PSRS.

program could use one Phases template with 5 phases. However, our implementation is consistent with the original description of the algorithm. It also provides an example of pattern template composition.

The two instances of the Distributor template are created and used by the two Phases templates. The first phase of `PSRSPhases` instantiates `PSRSDistributor` and uses it to provide the parallelism in the first phase. The third phase of `PSRSPhases` creates the second instance of the Distributor, `MergeDistributor`. This Distributor provides the parallelism for the third phase and the two subphases.

5.1.3 CO_2P_3S Solution

A screenshot of the CO_2P_3S solution, showing the two instances of the Distributor and Phases templates, is shown in Figure 5.3.

This program uses JGL, the Java Generic Library, in its implementation [79]. JGL provides a set of classes for different types of collections and generic algorithms for these collections. In particular, JGL provides array collections, ranges over a subset of an array, iterators over a given range, and a generic Quicksort algorithm for a given range. However, it was necessary to make one change to the library to implement PSRS. Operations on ranges and iterators in JGL synchronize on the underlying collection to ensure thread safety. In PSRS, this synchronization is unnecessary since each processor works on a disjoint range of the data. Unfortunately, this synchronization removes most of the parallelism as well, and was removed from the library.[1]

[1] A common problem when writing parallel Java code is excessive synchronization inside the standard class library, which reduces performance. This synchronization is included to help make the library thread–safe, which is desirable for most multi–threaded applications but not for parallel programs. The problem is exacerbated by the fact that the standard

This program takes advantage of the ability to add sequential methods to the parent and child classes in the Distributor framework. For example, this is used to access state in the children, such as the local samples generated in the first phase and to initialize the ranges in the children of `PSRSDistributor` before the first phase. It also takes advantage of the ability to encapsulate intermediate data from a program in the Phases frameworks. The samples and pivots are internal to the sort and should not be available with the results.

The implementation of the program is sketched out below. Of the six phases, the middle four phases implement the PSRS algorithm. The first and last phases, creating the data and verifying the sort, complete the application. For `PSRSPhases`, the list of phases and their implementation is:

1. `initializeData:` This phase creates an array of `Integer` objects to be sorted. Once the data is created, the framework for the first instance of the Distributor, `PSRSDistributor`, is instantiated. An added constructor in the parent class for this framework creates the ranges for the child objects and statically distributes these to the children using a sequential initialization method.

2. `sortPartitions:` Using a parallel method in the `PSRSDistributor` template, each child uses the Quicksort method from JGL to sort the array elements inside of its defined range. Once the array is sorted, each child takes a set of samples from its range, which are stored in an instance variable.

3. `getPivots:` A sequential method defined by the parent in `PSRSDistributor` collects and collates the complete array of samples from the children. The samples are sorted and sampled again to obtain pivot values. An array of pivot values is created, with markers (in this case, `null` objects) added to the start and end of the array.

4. `partitionData:` This phase partitions the data in the sorted sublists based on the pivot values. To do this, the `MergeDistributor` framework is instantiated and initialized in parallel. This initialization uses the neighbour distribution to partition a pair of consecutive pivots to each child. In parallel, each child obtains an array of ranges, one per sorted sublist in the children of `PSRSDistributor`, that are bounded by the two pivots. These ranges are computed by a sequential method added to the `PSRSDistributor` framework.

5. `mergeData:` This phase instantiates the `MergePhases` framework. The phases in this framework, described below, are executed.

documentation for Java classes does not indicate which methods are synchronized, and that a programmer cannot be sure that an unsynchronized method does not call a synchronized one in its implementation.

Table 5.1: Speedups and wall clock times for PSRS.

	Processors	2	4	8	16
12.5 million	Speedup	1.67	3.38	6.29	11.18
Integer objects	Time (sec)	531 ± 16	262 ± 6	141 ± 11	79 ± 2

6. `verifyResults`: This last phase verifies the sort using a parallel method defined in the `PSRSDistributor` framework.

The fifth phase in `PSRSPhases` implements the last phase of the PSRS algorithm. To do so, it creates and uses an instance of the second Phases template, `MergePhases`. The phases in this instance of the template and their implementations are:

1. `mergePartition`: In parallel, each child in the `MergeDistributor` framework creates a temporary buffer and merges the data in the ranges created in the fourth phase into that buffer.

2. `mergeFinalArray`: First, an array of offsets where each child should merge its temporary buffer into the original array is computed sequentially. A parallel method in the `MergeDistributor` framework distributes these offsets using a block distribution scheme. The child implementation of this method copies its temporary buffer into the original array based on the offset it receives.

5.1.4 Results

The performance results for PSRS, collected using the same execution environment as the reaction–diffusion example (see Section 4.3.1, under the "Performance" characteristic on page 99), are given in Table 5.1. These results are only for sorting; neither data initialization nor sort verification are included. The speedup is calculated with respect to the JGL Quicksort algorithm using the same data.

The choice of 12.5 million objects was dictated by memory constraints; this was the largest problem that could be run with a heap size of 512MB. Results with smaller data sizes showed a small drop in speedup as the number of processors increased, consistent with a loss in problem granularity, so only the single data point is presented.

In contrast to the reaction–diffusion application, this program scales well to 16 processors. The difference is that this program does much less synchronization; there are five synchronization points in the entire program, as opposed to two barriers per iteration of the reaction–diffusion computation (which takes hundreds to thousands of iterations to converge).

Because PSRS is an explicitly parallel algorithm, it took more effort to write this program. The only code that was used from the sequential version was the Quicksort from JGL, which is only used as part of the first phase. The code for the remainder of the first phase (segmenting the input array into sublists) and the rest of the phases had to be written specifically for the parallel version. Not including the Quicksort code, the complete PSRS program was 631 lines of Java code, of which 352 lines (56%) is user code. Once again, the structural code represents a large portion of the complete program. Again, though, this code is generated automatically.

5.1.5 Composing CO_2P_3S Frameworks

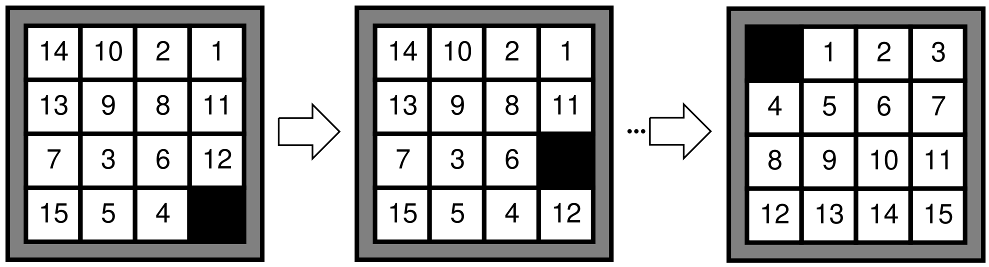
The PSRS example used a total of four design pattern templates. In our experience, many parallel programs need to use multiple patterns to be efficiently parallelized. Different parts of a parallel program can have different characteristics and requirements, which cannot usually be addressed by a single pattern.

In CO_2P_3S, a program that uses multiple pattern templates will generate a framework for each instance of each template. These frameworks must be composed into a single application. We explored some of the problems with this composition for general frameworks in Section 2.3.2. Again, this work does not address these problems directly, but they still need to be considered.

In Section 4.3.2, we noted that the frameworks generated by CO_2P_3S do not monopolize the flow of control of an application for its lifetime. Instead, the frameworks execute their computation (as dictated by the hook method implementations) and, when finished, return their results and program control to the user. The application code can now go on to do other work using these results. The framework may provide a single large computation, as in the Mesh template, or it may provide parallelism for a set of individual methods in a class, as in the Distributor template. Regardless, the call/return style of use allows the frameworks to be easily incorporated into an application.

In addition, CO_2P_3S frameworks use encapsulation to hide the internal objects in the frameworks. This encapsulation takes the form of a single class that allows the framework to be instantiated and used more easily. This class also serves as a single point of entry to a framework, which can be saved in an instance variable or passed as a parameter in method calls. Without this single entry point, the programmer would need to understand the internal details of the framework to know what objects are present and what responsibilities these objects have in the overall computation in order to correctly compose the frameworks. This encapsulation can also hide the parallelism in a framework. As much as possible, the use of parallelism is an attribute of the implementation of the object that is the point of entry to the framework. Ideally, collaborating objects should not need to be aware of the use of parallelism at all.

These characteristics of CO_2P_3S frameworks, encapsulation and the call/return style of use, allow the frameworks to be composed using normal object composition. Frameworks can be instantiated in user application code,



(a) An unsolved example of the 15–puzzle.

(b) A valid move, swapping a cell adjacent to the empty cell with the empty cell.

(c) The goal state for the 15–puzzle.

Figure 5.4: Examples of the 15–puzzle. The black square is empty.

either in the hook methods of other frameworks (as in the `mergeData` phase in `PSRSPhases`, which instantiated the `MergeDistributor` framework) or in auxiliary classes (such as the mainline class for the reaction–diffusion example, which instantiated the `RDMesh` framework). The framework objects can also be passed as parameters to method calls, which is used several times in the PSRS implementation. For instance, in the data partitioning phase, the initialization method for each child in the`MergeDistributor` framework is passed the instance of the `PSRSDistributor` framework class so the partitions can be computed.

5.2 Solving the 15–Puzzle Using Parallel Iterative–Deepening A* Search

5.2.1 Problem Description

The 15–puzzle is the most common instance of sliding tile puzzles. This puzzle consists of a set of 15 cells, each labeled with a unique value between 1 and 15, arranged on a 4×4 square grid with one empty cell. An example is given in Figure 5.4(a). To play, the player makes moves by selecting a labeled cell adjacent to the empty cell and swapping the two, as shown in Figure 5.4(b). The objective is to reach the goal state, in Figure 5.4(c), in the minimum number of moves.

To solve this problem, we used depth–first iterative–deepening A* (IDA*) search [58], which is based on A* search [47]. A* search maintains a list of unexplored nodes in a search and expands the most promising one. This is the node n with the lowest heuristic evaluation, based on the formula

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost of the path from the initial position to node n and $h(n)$ is the estimated cost from n to the closest goal. If $h(n)$ never overestimates the cost to a goal node (in heuristic search terminology, if $h(n)$ is *admissible*), then A* is guaranteed to find the solution with the lowest cost. The principle weakness of A* is the large amount of memory and processing time dedicated to managing the lists of unexplored and explored search nodes (needed to prevent duplicate searches). These two lists contain all nodes that are examined over the course of the search.

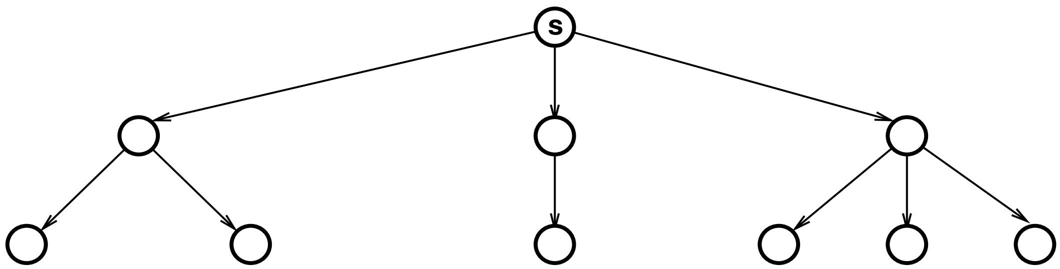
To address these problems, IDA* iteratively performs depth–first searches to find a solution. The search is controlled using a cost threshold t, initially set to $h(s)$ of the initial position s. IDA* searches to find a solution with cost t. Each branch is searched in a depth–first manner until either a solution is found or $h(n) > t$ for the current node n in the branch. Assuming that $h(n)$ is admissible, this condition means that there is no possible solution with cost t in the subtree rooted at node n. The current branch of the search tree is *pruned* and searched no further. If no solution is found (*i.e.* all branches have been pruned), t is incremented and the search is repeated with the new threshold. Again assuming that $h(n)$ is admissible, IDA* returns an optimal solution. An example of IDA* search, where the cost of a node is its depth in the tree, is given in Figure 5.5. The depth–first nature of IDA* reduces the amount of memory used as only the path from the root to the current node will be in memory.

In the 15–puzzle, $g(n)$ is the depth of the node in the search tree. A common heuristic function $h(n)$ is the sum of the Manhattan distances for each of the labeled tiles. The Manhattan distance for a labeled tile is the sum of the horizontal and vertical distances from the current position of the tile to its position in the goal.

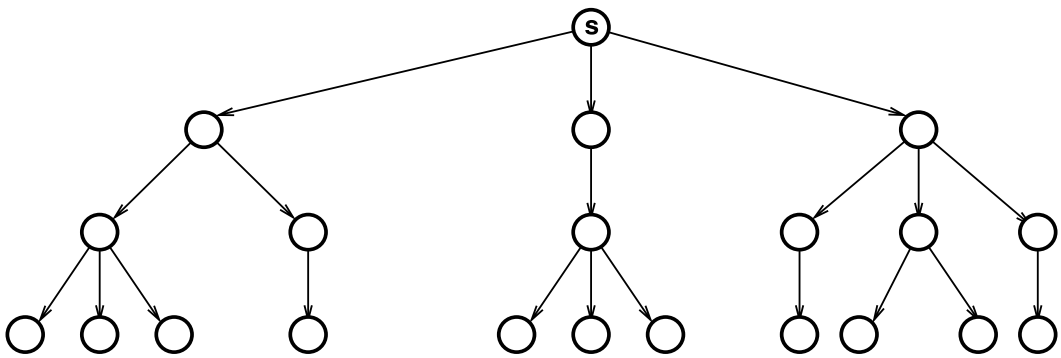
To parallelize this problem, we change the search algorithm to statically build the top part of the search tree, as shown in Figure 5.6. The root of the tree, in black, is the `InitialNode` or puzzle instance being solved. The interior nodes are `MasterNodes`, and fill in the positions in the interior of the subtree. The leaves of the subtree are instances of `FrontierNode`. To search this modified tree in parallel, a main thread traverses initial and master nodes in a depth–first manner. This traversal is used to compute the values of $g(n)$ and $h(n)$ and propagate them down the tree to the frontier nodes. The frontier nodes use the IDA* search algorithm on their position, using the $g(n)$ and $h(n)$ values from its parent.[2] Each frontier node search is independent, though, and can be assigned to another thread running on a different processor.[3] After the main thread has traversed the static subtree and distributed the frontier nodes

[2]The $h(n)$ value, or Manhattan distance, can be computed without any need for data from the parent. However, it is faster to use a table that computes the change in $h(n)$ based on the value from the parent and the move that results in the child node.

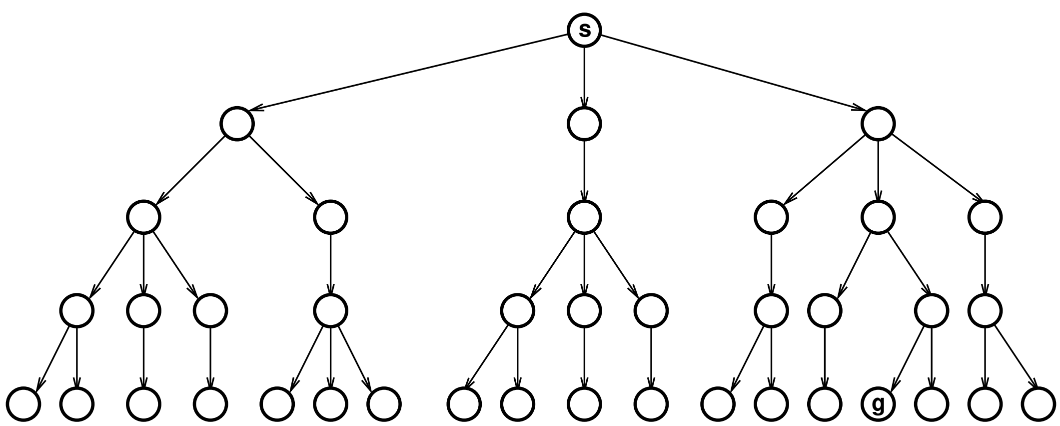
[3]This program does not have any search enhancements that could introduce any dependencies between the frontier nodes.



(a) Start by setting the depth threshold to $h(s)$, where s is the initial position at the root of the tree. For this example, the initial threshold $h(s)$ is 3. The search does not find a solution.



(b) Increment the depth threshold to 4 and search the tree again. There is still no solution.



(c) At depth 5, a solution is found.

Figure 5.5: A basic example of IDA* search, ignoring pruning. The solution, or *goal node*, is represented by node g.

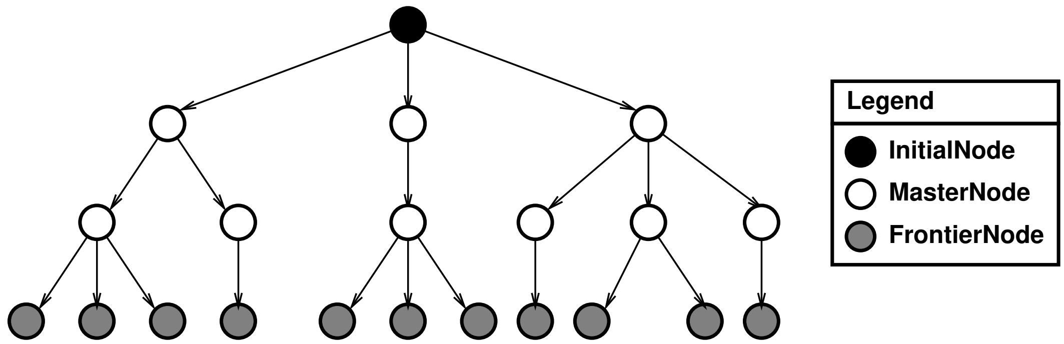


Figure 5.6: The structure of the static part of the IDA* tree.

to other processors, it waits for the results of the search. If a solution is found, the search stops. Otherwise, the cost threshold is increased and the search is repeated. Thus there is a synchronization point at each iteration.

The $g(n)$ and $h(n)$ values for the initial and master nodes never change. Once these values are propagated to the frontier nodes, there is no need to traverse the static subtree again. Instead, the search can iterate over the set of frontier nodes, assigning each to different processors and waiting for the results.

One of the biggest problems in parallel search is load imbalance. In a typical search, a small number of branches are responsible for most of the work. In the context of Figure 5.6, this means that a few of the frontier nodes may be expensive to search while the rest are quickly shown to have no solution [49]. To balance the workload across the processors, expensive frontier nodes can be *expanded* [17]. The frontier node is replaced with an equivalent master node, and its children are generated and added to the frontier.[4] Figure 5.7 shows an example where two of the frontier nodes from Figure 5.6 are expanded. Expanding the frontier in this way creates additional nodes that are less expensive to search individually. Partitioning this new frontier over a set of processors can result in better load balancing if the new frontier nodes, which may all be expensive to search, are distributed to different processors. To decide when to expand a frontier node, the time needed for the last search of the node can be kept. The node is expanded when this time exceeds a threshold.

Another important aspect of the frontier is its size. This is dictated by the size of the static part of the search tree. If the tree is too small, there will be few frontier nodes and load distribution may be skewed. With enough time, an expanding frontier will be able to correct this problem, but it may take several iterations to expand the frontier enough to balance the load. A deeper static tree generally provides a more balanced load.

Another common solution to the load balancing problem is to use *work–*

[4]To reduce the size of the search tree, a node will never generate a child node that is a duplicate of its parent. This requires that each child node keep a copy of its parent position. However, the static part of the search tree is sufficiently small that all of it is kept.

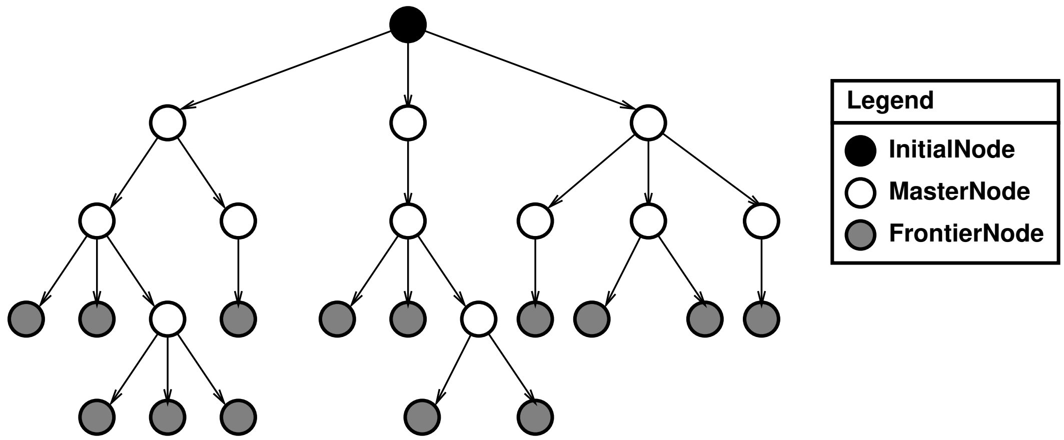


Figure 5.7: Expanding the frontier of the static tree in an IDA* search. Two frontier nodes from Figure 5.6 have been expanded.

stealing. When a processor runs out of work, rather than becoming idle it attempts to find and take outstanding work from another processor. Work–stealing is part of the basis for efficient fork/join implementations that rely on lightweight executable tasks, such as the framework by Lea [63] and Cilk [34]. It is also part of many parallel search algorithms, such as APHID [17].

Another problem with parallel IDA* search is that the first few iterations are usually inexpensive, typically searching only a few nodes. The parallel search should not be started until the iterations are expensive enough to warrant it. This decision can be made in two ways. First, the initial sequential iterations of IDA* can be timed. When this time becomes large enough, the parallel search completes the problem. Second, if an expanding frontier is used, the search can be done sequentially until a frontier node is expanded and is parallel thereafter. However, it may be necessary to lower the threshold for expansion for deeper static subtrees. Otherwise, it will take more iterations before any frontier node needs to be expanded, which delays the use of the parallel search and reduces overall performance.

To improve the overall performance of a parallel search, it is common to cancel the searches at all other processors once a solution is found. This can cause the last iteration of the search to finish quickly. In contrast, the time needed for the last iteration of a sequential search program can vary greatly, depending on the order in which the branches are traversed. This variation in both sequential and parallel search algorithms can cause the achieved speedups to vary considerably, and sometimes even results in superlinear benefits. This variance is normally addressed by examining a large sample of searches. With enough samples, the variance balances out and the results are representative of the expected performance of the program on a typical search.

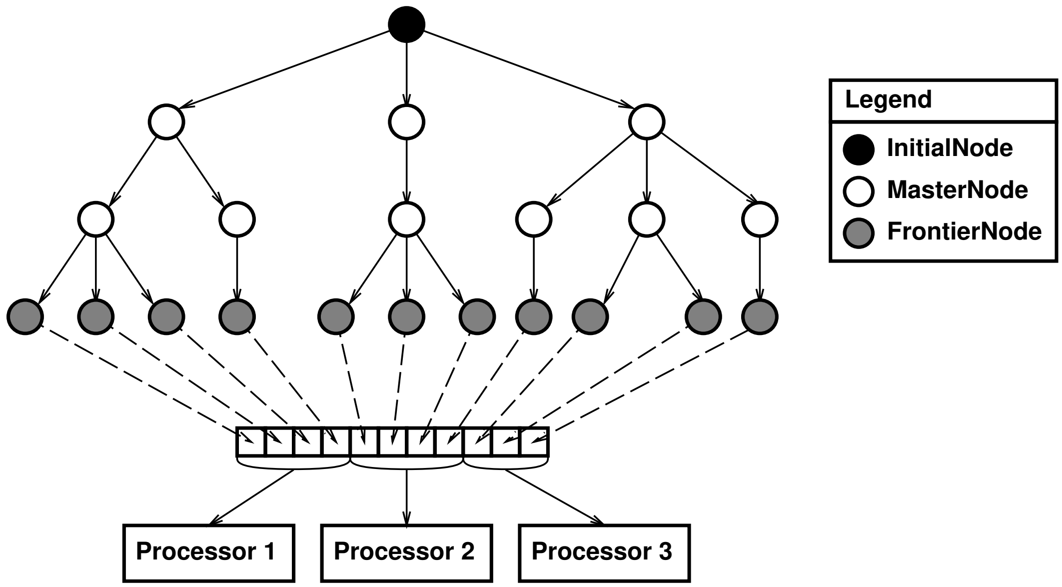


Figure 5.8: Parallelizing IDA* using the Distributor pattern template.

5.2.2 Pattern Selection

The Distributor pattern template can be used to parallelize this problem. The frontier nodes can be collected into an array, which can be distributed over a set of processors using a parallel method in the Distributor framework. Figure 5.8 illustrates this strategy.

To expand the frontier, the frontier nodes track the time for the last search. If the time for this search exceeds a threshold, the node is expanded. A new array of frontier nodes is collected and used for the next iteration of the search.

To effectively distribute work across the processors, the array of frontier nodes is distributed using a striped distribution. Frontier nodes in adjacent array positions will be distributed to different processors. This is important as the frontier nodes for a given branch of the tree will be in a contiguous portion of the array. This means that the expensive frontier nodes for the same branch will be clustered in a small region of the array. If the array were block distributed, there is a higher probability that these expensive frontier nodes would be assigned to the same processor and the load could become imbalanced.

Note that because the children of the Distributor template are independent, work–stealing is not used for this program. Such a change could, however, be made at the Intermediate Code Layer. A better solution would be to introduce a lightweight task design pattern template into CO_2P_3S.

5.2.3 CO_2P_3S Solution

A screenshot of the CO_2P_3S solution to the IDA* search problem is given in Figure 5.9.

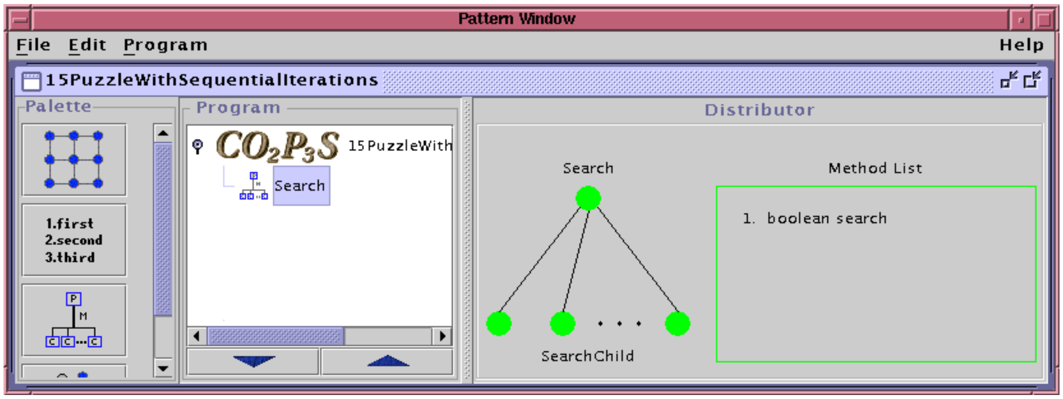


Figure 5.9: A screenshot of the CO_2P_3S implementation of IDA*.

The CO_2P_3S solution is straightforward. The single instance of the Distributor, `Search`, provides a method `search()` that takes an array of `FrontierNode` objects (stripe–distributed over the children) and the cost threshold for the current iteration. The children implement this method by iterating over the subset of nodes that are passed to them and execute a depth–first IDA* search on the positions to the given cost threshold. The return value is a Boolean indicating whether the solution has been found in the subset of frontier nodes.

The `search()` method in the `Search` instance of the Distributor will return an array of Booleans, one per child. To provide a better interface to the frontier search, the parent class in `Search` provides a sequential method `searchFrontier()`, which takes the same arguments as `search()`. This method calls the parallel `search()` method and then reduces the array of results into a single Boolean value indicating if the frontier search has found a solution. This code is shown in Figure 5.10. The child implementation of the `search()` method is given in Figure 5.11.

The complete IDA* program starts by building the static subtree and propagating the value for $g(n)$ and $h(n)$ to the frontier nodes. The frontier is then collected into an array. The search starts sequentially, iterating until at least one frontier node exceeds its expansion threshold. The frontier is expanded and the parallel search is started. The parallel search uses the `searchFrontier()` method in the `Search` framework for each iteration of IDA*, expanding and recollecting the frontier as necessary after each iteration. The parallel search is shown in Figure 5.12. The sequential search is similar but calls the `search()` method defined for the `FrontierNode` objects directly.

This program does not cancel the searches at other processors. The children in a Distributor are intended to be independent. As a result, all processors except the one that finds the solution search to the full threshold depth, which is the worst case for a parallel program. Simple strategies, such as a static completion flag, will help. In this implementation, we have chosen to maintain the independence of the children.

```
public boolean searchFrontier(FrontierNode[] frontier, int threshold)
{
    // Use the search() method defined for the Distributor to
    // distribute the frontier nodes and search them in parallel.
    boolean[] results = this.search(frontier, threshold) ;

    // Reduce the answer to a single true/false value.
    boolean foundAnswer = false ;
    int noChildren = this.getNoChildren() ;
    for(int i = 0;i < noChildren;++i) {
        if (results[i]) {
            foundAnswer = true ;
            break ;
        } /* if */
    } /* for */
    return(foundAnswer) ;
} /* searchFrontier */
```

Figure 5.10: The implementation of `searchFrontier()` in the `Search` class, which uses the parallel method `search()` (shown in Figure 5.11) and reduces the return value.

```
public boolean search(FrontierNode[] frontier, int threshold)
{
    int len = frontier.length ;
    boolean solved = false ;

    for(int i = 0;i < len;++i) {
        solved = frontier[i].search(threshold) ;
        if (solved) {
            break ;
        } /* if */
    } /* for */
    return(solved) ;
}    /* search */
```

Figure 5.11: The implementation of `search()` in the `SearchChild` class for the Distributor. This method searches the subset of nodes passed to it, which is the collected array of nodes on the frontier with a striped distribution applied to it.

```
public void parallelIterativeDeepening(int noChildren, int initialThreshold)
{
    boolean solved = false ;
    int threshold = initialThreshold ;
    FrontierNode[] frontier ;
    int frontierSize ;
    boolean recollectFrontier = false ;

    frontier = this.collectFrontierNodes() ;
    frontierSize = frontier.length ;

    // Create the Distributor instance for the parallel search.
    Search search = new Search(noChildren) ;

    // Don't go to the bottom, use the maximum depth.
    for(;threshold <= this._maxDepth;threshold += 2) {

        if (recollectFrontier) {
            frontier = this.collectFrontierNodes() ;
            frontierSize = frontier.length ;
            recollectFrontier = false ;
        } /* if */

        solved = search.searchFrontier(frontier, threshold) ;

        for(int i = 0;i < frontierSize;++i) {
            count += frontier[i].getNodeCount() ;
            if (frontier[i].shouldExpandFrontier()) {
                frontier[i].expandFrontier() ;
                recollectFrontier = true ;
            } /* if */
        } /* for */
    } /* for */
} /* parallelIterativeDeepening */
```

Figure 5.12: Parallel iterative deepening code from the `InitialNode` class. The initial sequential iterations are handled separately.

The synchronization point in this application is at the end of each iteration. An iteration of IDA* should only start after the previous iteration has finished, as the next iteration of the search cannot be canceled if it is started speculatively. This synchronization is implemented using the call/return style of use in the Distributor framework. The `searchFrontier()` method performs a single iteration of IDA* search, creating the necessary threads for the partitioned frontier. The method returns after all of the threads have completed their search of the partitioned frontier. The application code iteratively calls this framework method with a growing threshold parameter.

To better evaluate the performance of the Distributor pattern, the search is cut off before the last iteration is started. This avoids the variance in the search at the last iteration of IDA*.

5.2.4 Results

The performance results for this program were collected on an SGI Origin 2100 with 4 350MHz R12000 processors and 1 GB of memory. The program was run on the Java 1.3 VM using native threads, and compared to a sequential program run with green threads. The sequential program does not use the static tree, but rather performs a recursive depth–first search on the puzzle positions.

Both the depth of the static part of the search tree and the time threshold for frontier node expansion are parameters to the program. The program was run with the static tree depth set to three, five, and seven. The expansion thresholds used were 250ms and 500ms. The program was run on a set of 100 problems [58], which have sequential search time varying from a 0.1 seconds to 84 minutes. The reported speedups, given in Table 5.2, are based on a sample of problems that show the tradeoffs between both the threshold and the depth of the static tree.

Problems 47 and 48 (Tables 5.3(a) and 5.3(b)) are two moderately sized problems, running in about 33 seconds each. These two problems show that, for smaller problems, the combination of frontier node expansion threshold and static tree depth can have a large impact on the performance. The number of parallel iterations of IDA* are affected. For example, with a 250ms threshold for both problems, the last two iterations of IDA* are parallel with a static tree depth of three, but deeper trees only execute the last iteration in parallel. In most cases, the extra parallelism provides substantially better results. With a 500ms threshold, the depth of the tree determines whether any iterations of the search are parallel; with a static tree of depth seven, none of the frontier nodes are expanded so the parallel search is never triggered.

These two problems also help show the dynamic nature of search algorithms. For example, in Problem 48, the two worker case with a 250ms threshold exhibits only a small drop in performance when the depth of the static tree increases from three to five, despite the loss of a parallel iteration. The last parallel iteration at depth three tends to be unbalanced, whereas

Table 5.2: Speedups and wall clock times (in seconds) for parallel IDA* search for selected problems, with different static tree depths and frontier expansion thresholds. The number of parallel iterations, always at the end of the search, is also provided.

	Workers	2	4	8
Depth 3	Time	17.35	11.37	10.85
250 ms	Speedup	1.9	2.9	3.04
	Par. Iterations	2	2	2
Depth 5	Time	19.87	14.17	13.36
250 ms	Speedup	1.66	2.33	2.47
	Par. Iterations	1	1	1
Depth 7	Time	20.19	13.43	13.25
250 ms	Speedup	1.63	2.45	2.49
	Par. Iterations	1	1	1
Depth 3	Time	19.36	13.58	13.07
500 ms	Speedup	1.7	2.43	2.52
	Par. Iterations	1	1	1
Depth 5	Time	19.63	14.48	15.74
500 ms	Speedup	1.68	2.27	2.09
	Par. Iterations	1	1	1
Depth 7	Time	32.4	32.5	32.96
500 ms	Speedup	1.02	1.01	1
	Par. Iterations	0	0	0

(a) Problem Number 47 with four processors.

	Workers	2	4	8
Depth 3	Time	19.55	12.86	10.53
250 ms	Speedup	1.76	2.68	3.28
	Par. Iterations	2	2	2
Depth 5	Time	21.07	15.64	14.1
250 ms	Speedup	1.64	2.21	2.45
	Par. Iterations	1	1	1
Depth 7	Time	20.81	13.44	13.9
250 ms	Speedup	1.66	2.57	2.48
	Par. Iterations	1	1	1
Depth 3	Time	23.78	17.54	15.12
500 ms	Speedup	1.45	1.97	2.28
	Par. Iterations	1	1	1
Depth 5	Time	22.17	15.87	14.89
500 ms	Speedup	1.56	2.17	2.32
	Par. Iterations	1	1	1
Depth 7	Time	33.81	33.91	34.41
500 ms	Speedup	1.02	1.02	1
	Par. Iterations	0	0	0

(b) Problem Number 48 with four processors.

	Workers	2	4	8
Depth 3 250 ms	Time	611	347	311
	Speedup	1.89	3.32	3.7
	Par. Iterations	4	4	4
Depth 5 250 ms	Time	601	309	300
	Speedup	1.92	3.74	3.85
	Par. Iterations	3	3	3
Depth 7 250 ms	Time	611	321	315
	Speedup	1.89	3.59	3.66
	Par. Iterations	3	3	3
Depth 3 500 ms	Time	606	327	336
	Speedup	1.9	3.52	3.43
	Par. Iterations	3	3	3
Depth 5 500 ms	Time	584	340	318
	Speedup	1.97	3.39	3.63
	Par. Iterations	3	3	3
Depth 7 500 ms	Time	610	339	345
	Speedup	1.89	3.4	3.34
	Par. Iterations	2	2	2

(c) Problem Number 96 with four processors.

	Workers	2	4	8
Depth 3 250 ms	Time	2696	1445	1446
	Speedup	1.88	3.49	3.51
	Par. Iterations	4	4	4
Depth 5 250 ms	Time	2657	1425	1322
	Speedup	1.91	3.56	3.84
	Par. Iterations	4	4	4
Depth 7 250 ms	Time	2642	1362	1365
	Speedup	1.92	3.72	3.72
	Par. Iterations	3	3	3
Depth 3 500 ms	Time	3031	1625	1399
	Speedup	1.67	3.12	3.63
	Par. Iterations	4	4	4
Depth 5 500 ms	Time	2757	1466	1393
	Speedup	1.84	3.46	3.64
	Par. Iterations	3	3	3
Depth 7 500 ms	Time	2675	1416	1361
	Speedup	1.9	3.58	3.73
	Par. Iterations	3	3	3

(d) Problem Number 100 with four processors.

the only parallel iteration at depth five is much more balanced. Combined with the fact that the first parallel iteration at depth three is still relatively inexpensive, taking about three seconds, it becomes easier to see why so little performance is lost. In contrast, with four or eight workers, the load balancing for the last parallel iteration is about equal, so the extra parallel iteration with the static tree of depth three provides better performance.

These load balancing issues appear in other tests. We expect that as the depth of the static tree increases, with the same number of parallel iterations, the performance should increase. A deeper static tree increases the number of frontier nodes and reduces the amount of computation for each node. On average, distributing these nodes over a set of threads should yield better load balancing. In Problem 48 this is the case. For those problems with identical numbers of parallel iterations, performance uniformly improves with a deeper static tree. Problem 47, however, does not exhibit this characteristic, particularly with the larger threshold value. The load balancing becomes skewed and performance suffers.

This load imbalance experienced in Problem 47 may be a result of a bad distribution of frontier nodes. Unlike the workpile model, which allocates work to processors based on demand, the Distributor template statically distributes the array of frontier nodes to processors. The workpile model is better suited to the dynamic nature of search algorithms [17].

The two remaining problems, 96 and 100 (Tables 5.3(c) and 5.3(d)), represent two of the best results on two of the largest problems. These two examples highlight two points. First, they show the upper bound on the achievable speedups for this problem. In spite of the potential for load imbalance due to static frontier node distribution, these large problems show that near–linear speedups are still achievable, noting that the problems were run on a four–processor machine. Many of the speedups exhibited by the two problems speedups are within 10% of ideal. Second, the two problems also highlight the dynamic nature of search. A rule of thumb in parallel programming is that larger problems on the same number of processors will improve performance. From the two tables there are several cases where the smaller program achieves better performance.

The complete parallel IDA* search program was 486 lines of code. 393 lines (81%) of this is user code for the problem. The parallel framework code consists of a single instance of the Distributor with a single parallel method, which is 93 lines of code. Of the 393 lines of user code, 117 (30%) were taken from the sequential version of the program (which was 188 lines of code itself). The extra code in the parallel version is needed to create and manipulate the static subtree. The sequential program, on the other hand, consists mainly of a single recursive method that does a depth–first search on a position, which is reused in the sequential version to search the frontier nodes. In addition, both versions of the program have a table to compute the incremental change in Manhattan distance, a table that has precomputed all of the possible placements of the blank tile in the children of a given position,

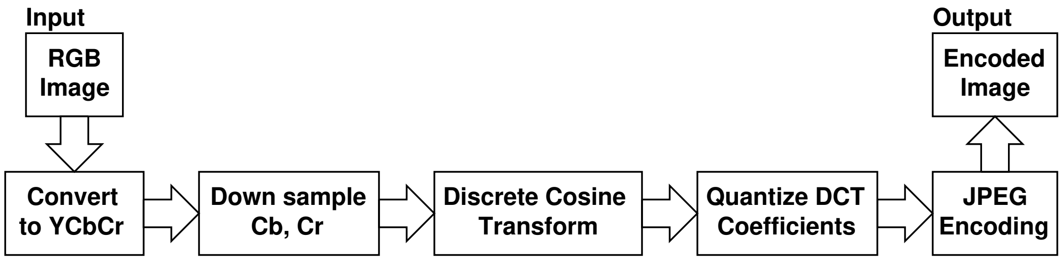


Figure 5.13: The steps in JPEG compression.

and a class to parse the input file with the 100 problems.

5.3 JPEG Compression

5.3.1 Problem Description

The last application program in this chapter is parallel JPEG compression. This program takes an RGB image, read from a GIF image using classes in the Java standard library, compresses it using the baseline JPEG standard, and saves the result in the JPEG File Interchange Format (JFIF) [45].[5]

The compression process consists of six steps, illustrated in Figure 5.13. These stages are:

1. Convert RGB input to YCbCr. JPEG compression does not work on RGB images, but rather works in a luminance/chrominance colour space. The first step converts the RGB data to one such space, YCbCr.

2. Downsample. JPEG is *lossy* compression; the compression is achieved by selectively filtering out information about the image and not storing it, so the resulting JPEG image is not a perfect reproduction of the original image.[6] The human visual system can tolerate a certain amount of degradation before an image appears distorted. In a luminance/chrominance colour space, the visual system can tolerate the loss of chrominance information. This stage, which is optional in JPEG compression, downsamples the Cb and Cr components of the image from the previous step. The downsampling takes an $N \times N$ component and produces an $\frac{N}{2} \times \frac{N}{2}$ output where each element is the average of a 2×2 portion of the original. The luminance component, Y, is not downsampled.

[5] The JPEG standard, now over 10 years old, specified how to compress and encode image data but did not specify how the image and associated information was to be stored. JFIF is still the defacto standard in spite of the creation of the SPIFF file format standard by the JPEG Working Group in 1997 [52].

[6] There is a version of the JPEG standard for lossless compression for images that cannot tolerate any degradation, such as medical images.

3. Discrete Cosine Transform (DCT). The Discrete Cosine Transform converts from the spatial domain to the frequency domain. The pixel values are normalized to the range $[-127, 128]$ and the transform is done on each 8×8 pixel subimage of each of the Y, Cb, and Cr components of the down sampled image.[7] The two–dimensional DCT replaces each 8×8 subimage with an 8×8 array of DCT coefficients. The first coefficient, at location $(0, 0)$, is called the DC coefficient. The remaining 63 are the AC coefficients.

4. Quantize the DCT coefficients. Quantization provides the compression in the JPEG standard. This process filters the DCT frequency coefficients using a *quantization table*. The quantization table is an 8×8 table of factors that are applied to the corresponding 8×8 array of DCT coefficients, illustrated in Figure 5.14. This table determines the accuracy with which each of the frequency coefficients should be preserved. The smaller the value in the quantization table, the better preserved that frequency will be in the compressed image. Higher values preserve less information and provide better compression at the expense of image quality. In many image compression programs, the quality metric for a JPEG image refers to a factor that is applied to each element of the quantization table.

5. JPEG encoding. After an 8×8 subimage has been quantized, it is encoded as a stream of bits. The zero values are encoded very compactly, compressing the image. The encoding uses variable length Huffman encoding. The AC and DC components are encoded differently and use their own separate Huffman codes. The DC component is encoded as the difference between the DC component of the current subimage and the DC component of the previous subimage. The 63 AC components are encoded using a zig–zag pattern that encodes the frequency coefficients that are most important, near the top left corner, first. The less important higher frequencies, which have a high probability of being filtered out and thus do not need to be stored, are encoded last.

   The complete image is encoded by appending the encodings for each subimage for the Y, Cb, and Cr components in the order illustrated in Figure 5.15. Recall that the Cb and Cr components were down sampled in the second stage. The encoding for each 8×8 subimage is the encoded DC component followed by the encoded AC components.

[7]Images with dimensions that are not multiples of eight (or, rather, 16 to account for the down sampling) are padded, and the actual dimensions of the image, stored in the JFIF file, are used to crop the extra data.

236	−11	12	−5	2	−2	−3	1
−23	18	6	−3	−3	0	0	−1
−11	9	−2	2	0	−1	−1	0
−7	−1	0	0	2	1	0	0
−1	0	2	2	−1	1	−1	2
2	0	2	0	−1	2	1	−1
−1	1	0	−2	−1	2	1	−1
−3	2	−4	−2	2	−1	1	−2

(a) An 8×8 array of DCT coefficients.

$$\Rightarrow Q(i,j) = \frac{DCT(i,j)}{QT(i,j)} \Rightarrow$$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(b) An example quantization table.

15	−1	1	0	0	0	0	0
−2	2	0	0	0	0	0	0
−1	1	0	0	0	0	0	0
−1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(c) Resulting quantized coefficients, rounded to the nearest integer.

Figure 5.14: An example of the quantization process on an 8 array of normalized DCT coefficients.

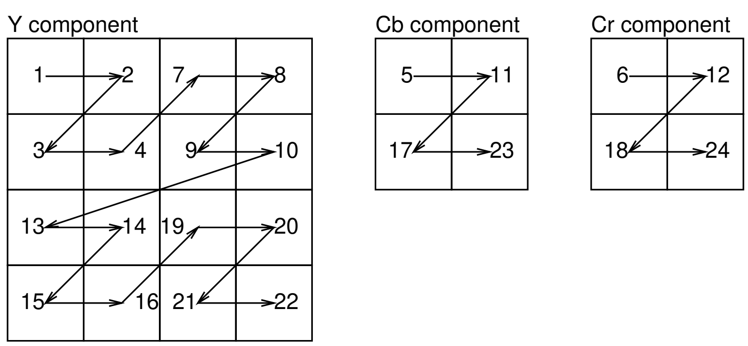


Figure 5.15: The order in which the encodings for the subimages are appended together into a complete JPEG encoding. Each block is an 8×8 subimage.

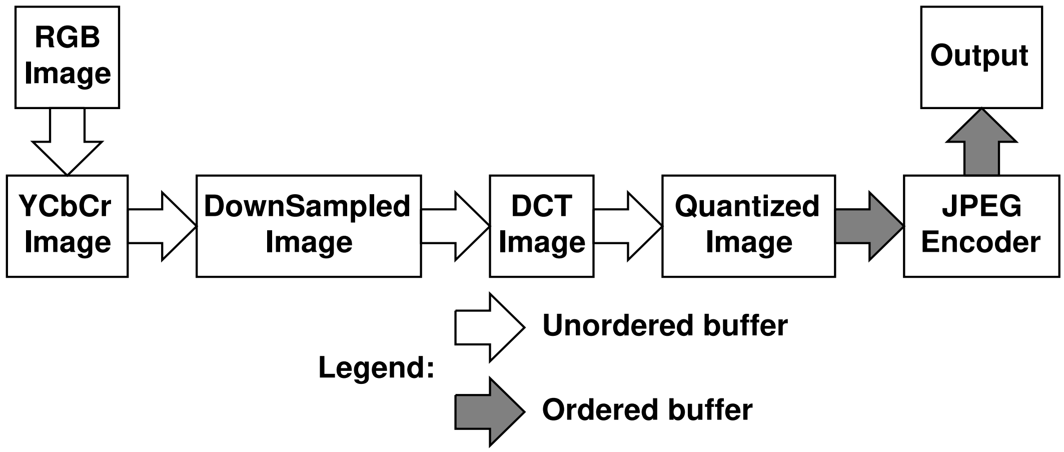


Figure 5.16: The pipeline specification for JPEG compression, with an amalgamated encoding stage.

5.3.2 Pattern Selection

With the exception of the encoding process, each of the steps in JPEG compression can be executed independently on any 16×16 subimage of the original input image. The DCT and JPEG encoding are performed independently on 8×8 subimages, but the second stage takes the 16×16 Cb and Cr components and downsamples them to 8×8 components. Further, no ordering is needed between the subimages until the encoding and output stages. Using the Work–pile–based pipeline, the program can be implemented using the pipeline in Figure 5.16.

It is important to understand how the ordered buffers in Figure 5.16 operate. An ordered buffer does not preclude the stages at both ends of the buffer from executing concurrently. One part of an image can be quantized while another part is simultaneously being encoded while yet a different output result is being processed. An ordered buffer simply ensures that the work items exit in the sequence, regardless of the order in which they enter the buffer. The buffer tracks the sequence number of the work items that have been processed and only allows the next item to be removed when the previous item has finished processing. For instance, the encoding stage will always receive the subimages in the correct order. Note too that this ordering does not affect the operation of the stage that is placing data into the buffer. For example, the quantization process can still process work items out of order, and can be busy even if the encoding process is waiting for the next item in the sequence.

It is also possible to split the encoding process into two stages, one for encoding the AC components and one for encoding the DC components. The AC component of each of the subimages can be encoded independently. Only the DC component has a dependency, with the DC component of the previous subimage. This revised pipeline is given in Figure 5.17. While splitting the encoding creates more parallelism by adding an extra stage, it requires extra

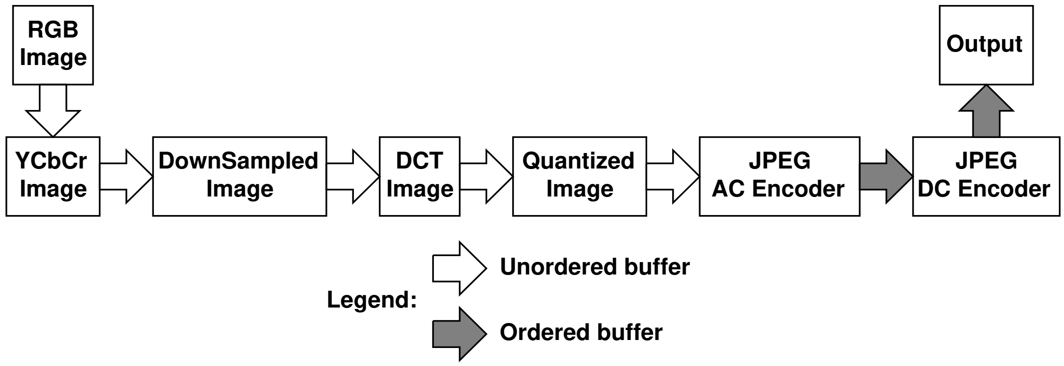


Figure 5.17: An alternate pipeline for JPEG compression, with the encoding stage split into two separate stages.

copying of the encoded subimages to create the final bit stream of JPEG data. This occurs regardless of the order in which the encoding stages are placed in the pipeline.

5.3.3 CO_2P_3S Solution

Since the Work–pile–based pipeline has not yet been implemented in CO_2P_3S, these results are based on a hand–coded implementation of a possible pipeline framework. The framework implements the Work–pile–based pipeline described in Section 4.2.3. The ordered buffers are implemented using the concurrent priority queue from the `util.concurrent` class library [62], with the ordering based on a work item tag that is assigned to the stream of input items as they enter the pipe and is maintained by each subsequent stage. In addition, a stage with an ordered input buffer has a reference to the last work item that was executed. As a result, the execution of ordered stages is guarded by a mutex lock to ensure that no work item is processed before the previous one has finished. Stages with no ordering between them have no buffers at all; the next stage is simply executed by the current thread until ordering becomes necessary.

Regardless of which of the two pipeline solutions is used, there are two ordered stages. The first ordering point is before the DC component is encoded, either in a separate pipe stage or as part of the complete encoding process. The output stage is also ordered so the mainline program will receive the output items in the correct sequence. This is needed to correctly assemble the bit streams of encoded JPEG data into the final encoded image. Since variable length encoding is used, this dependency cannot be avoided.

The input to the pipeline is contiguous stripes of 16 rows of the input image. If the number of rows in a stripe is not a multiple of 16, some stages (notably the downsampling, DCT transform, and AC encoding) will require data from other work items (see Section 5.3.2 for details). The work items in the pipeline would no longer be independent, so ordering buffers would be required between every stage and performance would suffer. To keep the program simple, stripes of 16 rows are used. It is also possible to partition each stripe into blocks. Again, this is easiest if the number of columns in these blocks is a multiple of 16.

One of the potential problems with the Work–pile–based pipeline, noted in Section 4.2.3, is contention for the heap lock when creating objects for each request for each stage. Through profiling, we were able to verify that this is indeed a problem in the pipeline framework. To reduce contention, the JPEG compression program was written to reuse as much storage as possible by performing each stage in place on the image data. To further cut down on contention, and allow us to test our pipeline rather than the ability of the virtual machine to create objects, the memory for each stage object was preallocated. These objects were stored in a memory pool that was accessed by the work item tag, so the preallocated objects could be retrieved without any locking. It may be possible for a pipeline framework to include similar preallocated object pools.

Both variations of the encoding stage were implemented. The differences between the two versions are restricted to three classes:

1. a class internal to the framework that is responsible for creating the pipeline structure (namely the buffers between the ordered stages). This class needs to be generated with the pattern template parameters that indicate the classes and ordering in the pipeline, so this change was expected. This class also provides an accessor to the stream of output objects for the pipeline results. In this framework, these objects are instances of the last stage, which will be one of the two encoding stages. The return type of this method also had to be changed.

2. the `QuantizedImage` class, which had to be changed to create the desired encoding stage as the next stage in the pipeline.

3. the mainline class. The mainline class reads the stream of output objects from the pipeline. This class had to be changed to read the correct output type.

The fact that changing the structure of the pipeline has such localized effects on the framework suggests a good separation of concerns and encapsulation. For example, the threads executing the pipeline do not need to explicitly account for any changes in the pipeline stages; this is handled by the polymorphic calls to the stage objects based on methods defined by an abstract superclass. Also, the search for work items iterates over a collection of buffers between stages. While the contents of this collection will change with different pipelines, the search itself is the same.

One potential optimization for the pipeline was to insert unordered buffers into the pipeline to create more work items. This optimization was applied, adding an unordered buffer between the down sampling and DCT transform. The performance of the program was unaffected.

In this program, the worker threads search for work from the last stage and work backwards towards the input queue. Since the only buffers in the program are just before the encoding stage and the input buffer, this policy gives a higher priority to encoding image data than to processing new work items. Scheduling work from the input queue forward can cause many work items to queue up before the DC encoding stage, which can only be executed by a single thread at a time. The remaining threads would be idle, reducing the performance of the program.

As expected in the Work–pile–based pipe, changing the number of threads for the different experiments requires no change to the structure of the pipeline. More worker threads are started, but these new threads can process work for the stages as it becomes available. The structure of the stages does not need to be changed to accommodate the new threads. The number of worker threads is an argument to the constructor for the pipeline framework.

5.3.4 Results

The speedups and wall clock times for the parallel JPEG compression application are given in Table 5.3 for JDK 1.2 and Table 5.4 for JDK 1.3. Results are included for both implementations of the encoder (the amalgamated encoding stage and the split encoding stage) using the Work–pile–based pipeline. The programs were executed on a Sun server with four 450MHz Sparc processors and 4GB of memory.

In addition, results are provided for the traditional pipeline from Figure 4.18 (page 87). Again, there are two implementations for this pipe, with an amalgamated encoding stage and with a two split encoding stages. Each stage in the pipe is assigned to a separate thread. This requires six worker threads for the amalgamated encoding solution and seven worker threads for the split encoding solution.

The times in the tables include both JPEG encoding and writing the resulting JFIF file to disk. Reading the GIF file and memory preallocation are not included. The number of threads listed is the number of workers created by the pipeline, which does not include the main thread. This thread is still active, but is limited to getting results from the output queue of the pipeline and does little computation.

To provide fair results, the sequential program was changed to preallocate its heap storage as well. Although there is no heap lock contention, the creation of objects still takes processing time. Including this time in the sequential code would skew the results.

The first trend that is apparent from the results is that the addition of new threads to a program using a Work–pile–based pipeline almost always improves its performance. The cases where performance suffers are when the number of threads is greater than the number of processors in JDK 1.3. Since the number of threads in the Work–pile–based pipe is independent of the stages of the pipeline, though, it is straightforward to find and use the optimal number of

Table 5.3: Speedups and wall clock times (in milliseconds) for the parallel JPEG encoder with JDK version 1.2.

	Workers	2	4	6	8
1024× 896	Speedup	1.64	2.31	2.41	2.72
	Time	2286±18	1626±13	1555±39	1381±29
3212× 3600	Speedup	1.7	2.56	3.08	3.47
	Time	30204±320	20117±194	16691±258	14831±97

(a) Results with an amalgamated encoding stage in a Work–pile–based pipeline with four processors.

	Workers	2	4	6	8
1024× 896	Speedup	1.6	2.21	2.52	2.81
	Time	2338±42	1697±57	1488±47	1335±20
3212× 3600	Speedup	1.14	1.52	1.72	1.85
	Time	45312±479	33754±265	29845±192	27798±163

(b) Results with the encoding split over two stages in a Work–pile–based pipeline with four processors.

	Encoder	Amalgamated encoding	Split encoding
1024 × 896 image	Speedup	1.97	1.92
	Time	1902±41	1959±32
3212 × 3600 image	Speedup	2.04	1.3
	Time	25270±371	39622±128

(c) Results using a traditional pipeline with six stages executing on four processors.

Table 5.4: Speedups and wall clock times (in milliseconds) for the parallel JPEG encoder with JDK version 1.3.

	Workers	2	4	6	8
1024× 896	Speedup	1.63	2.24	2.03	1.65
	Time	4358±84	3163±88	3499±80	4300±141
3212× 3600	Speedup	1.71	2.87	3.14	3.12
	Time	45512±300	27046±179	24744±249	24905±450

(a) Results with an amalgamated encoding stage in a Work–pile–based pipeline with four processors.

	Workers	2	4	6	8
1024× 896	Speedup	1.83	2.5	2.51	2.35
	Time	3872±45	2831±73	2826±42	3012±46
3212× 3600	Speedup	1.34	1.97	2.17	2.22
	Time	57914±538	39391±223	35709±275	35004±317

(b) Results with the encoding split over two stages in a Work–pile–based pipeline with four processors.

	Encoder	Amalgamated encoding	Split encoding
1024 × 896 image	Speedup	2.0	2.16
	Time (msec)	3547±29	3275±64
3212 × 3600 image	Speedup	1.86	1.46
	Time (msec)	41750±333	53324±388

(c) Results using a traditional pipeline with seven stages executing on four processors.

worker threads. In contrast, the number of threads in the traditional pipeline is a function of the structure of the stages, which would need to be adjusted. In addition, the improved performance obtained with the Work–pile–based pipe over the traditional pipe, even with an equivalent number of threads, shows that the threads are being used more effectively in our new formulation.

Another trend is that the amalgamated encoding solution yields better results as the image size increases. Even though the amalgamated encoding solution restricts concurrency (as the encoding for a complete stripe of the input image is done by a single thread at a time), the extra copying in the split encoding solution turns out to be more expensive. For small images, though, the extra concurrent stage present in the split encoding solution outweighs the extra copying. Using JDK 1.3, the split encoding pipeline outperforms the amalgamated encoding pipe. With JDK 1.2, the difference between the two versions is small.

One further trend is that the execution times for JDK 1.3 are larger. We have not found an explanation for this behaviour.

In terms of the implementation, parallel JPEG compression with an amalgamated encoding stage totals 1149 lines of Java code, where 911 lines (79%) of this is user code. For the pipeline with the split encoding, the program is 1256 lines of code, with 1016 lines (81%) of user code. For reference, the sequential program totaled 708 lines of code. The principle change between the two versions was that each stage of the compression process had to be modified to work with a subimage. This was simplified by the use of a bounded image in the sequential version, which operates exactly like the bounded arrays in the Mesh framework. In addition, the parallel version has classes for some stages that were not implemented as classes in the sequential version.

5.4 Summary

This chapter examined three additional applications developed with the CO_2P_3S parallel programming system. These three applications, sorting, IDA* search, and JPEG compression, come from different problem domains. Each design pattern template currently supported by CO_2P_3S was used in at least one of these problems. The sorting example was also noteworthy in that it used four pattern templates and composed the generated frameworks into a complete application. These applications help demonstrate the *utility* of both CO_2P_3S and the pattern templates it supports.

Chapter 6

Assessing the Usability of CO_2P_3S

Many tools are built and used by the same group of researchers. All usability claims about these tools are the product of anecdotal evidence from the system developers, and the possibility of inadvertent bias cannot be discounted. Because the system developers are so familiar with their tool, most usability information must be considered as the best case. This includes the information derived from the example programs written by the author of this dissertation. For instance, the developers of a system understand the inner workings of their tool and may use that knowledge when tuning an application. The resulting program may obtain better performance than a normal user could expect. As well, the developers will better understand the programming model provided by their system and thus be able to write programs more quickly than other users. For example, new CO_2P_3S users have to learn how to use the design pattern template parameters to specialize a given template for a program and how to use the hook methods in the generated frameworks to build an application. As the creator of the pattern templates and frameworks, the author did not suffer from this learning curve.

To assess the usability of CO_2P_3S, we conducted a usability study modeled after the studies for Enterprise [99, 96]. The study compared parallel programming with the CO_2P_3S parallel programming system against programming in Java using threads. The subjects for the study are a group of novice parallel programmers from an undergraduate class on object–oriented programming languages. This chapter describes the study and its results.

Section 6.1 lists some aspects of usability and discusses its importance. The design of the study is described in Section 6.2. Section 6.3 presents the results of the study, derived from application code metrics. Over the course of the study, we encountered several problems that limited the data we were able to collect. Section 6.4 describes these problems. Finally, Section 6.5 concludes this chapter with suggestions for future studies, to obtain more data and to correct some of the errors in this study.

6.1 The Importance of Being Usable[1]

Despite its importance, the usability of the tools produced in parallel programming systems research is rarely considered. Usability covers many issues, including (but not limited to):

- The learning curve for the system. A system should be easy to learn for both novice users and experienced parallel programmers.
- The effort required to create a working program using the system.
- Compatibility with existing software. Legacy code cannot be ignored. A parallel programming system should allow a user to integrate existing code into an application.
- The probability of programming errors. Some systems, such as Enterprise and CO_2P_3S, use a combination of user interface and programming model to reduce this probability. Other systems, most notably message–passing libraries, opt for flexibility at the expense of increasing the probability of user error.
- The performance of programs created with the system. A parallel programming tool that cannot produce applications with improved performance is of no use to programmers.
- The functionality of the system. The system must provide access to basic development facilities, such as a compiler and an editor.
- The toolset available in the system. Even though the abstractions in the programming model of a parallel programming system reduce the development effort, tools are still needed for debugging and performance tuning.
- How useful the system is for both novice users and experts. Ideally we would like a system to meet the needs of both groups of users. With both groups using the same tools, they can collaborate more effectively and novices can more easily graduate to experts.
- The ability to perform incremental tuning. Users should be able to concentrate their tuning efforts on just those parts of a program that are causing performance problems, without having to consider the complete application structure.
- Suitability for large–scale software engineering. Most parallel programming systems use several small– to medium–sized programs to demonstrate its features. The problems that real users want to solve can be

[1] **With apologies to Oscar Wilde.**

much larger in scope, involving months or years of development time. A system should be able to support large projects.

It is critical that the usability of a system be assessed with respect to end users and not its developers. The primary reason, which is surprisingly easy to overlook, is that the goal of parallel programming systems research is to create tools that other programmers can use to solve their problems. It is the needs of these programmers that must be considered as researchers create new abstractions and tools.

With the exception of the studies discussed in Section 2.1.3 (under the *usability* characteristic), we are aware of no other research that assesses the experiences of real users with a parallel programming system. These assessments will be necessary if parallel programming systems developers hope to produce the kinds of systems that programmers will use. Although system developers have preconceptions about the benefits their systems offer to users, usability studies almost invariably uncover additional benefits and weaknesses that were not anticipated. Uncovering this data will be crucial if the next generation of parallel programming systems are to succeed.

6.2 Study Setup

The best way to evaluate the usability of a parallel programming system is to run a controlled experiment. Subjects are split into two groups, and each are given a set of problems to solve. One group uses the parallel programming system and the other does not. The experiences of both groups are compared to determine the strengths and weaknesses of the tested system. However, preparing and executing a study to quantitatively assess the usability of a tool is not a simple matter, which is why few research groups have attempted it. The experimenters must determine what measurements they wish to take and find ways to take them, which involves instrumenting the programming environment for both groups of subjects. The experiment should be held in a controlled environment so that correct measurements can be taken, which takes time to construct. The problems for subjects to solve must be chosen carefully. The problems cannot be too small or too simple or they will not be representative of actual program development. In addition, the problems should cover a range of different parallel programming styles to evaluate the *utility* of the system.

This section describes the design of a study to assess the usability of CO_2P_3S. Since we were unable to control the environment, it cannot be said that we conducted an experiment. All materials used in the study are given in Appendices C through F. Appendix C is the documentation for the Mesh pattern template that was supplied to the subjects. Appendix D provides the assignment descriptions used for the study. Appendices E and F are background material on sequential and parallel mesh computations respectively, that were also distributed to the subjects.

6.2.1 Design of the Usability Study

The subjects for the study were undergraduate students enrolled in CMPUT 425, a course on implementation issues in object–oriented programming languages. There were 20 students in this class, which were broken into two groups of 10.

The study consisted of two smaller studies. In the first study, both groups implemented a variation on the LaPlace solver. One group used the Patterns Layer of the CO_2P_3S parallel programming system. The other group used non–CO_2P_3S Java with native threads, and were given an implementation of barrier synchronization that they could use in their parallel program.

In the second study, the two groups switched programming environments. The problem was the same reaction–diffusion example used in Chapter 3.[2]

Both of the problems used in this study are examples of mesh computations. The subjects were given background material on parallel and sequential versions of mesh problems. They were also instructed to consider both performance and object–oriented design in their assignments. In particular, they were encouraged to attempt to maintain some separation between the application–independent structure and the application–specific computation.

Each subject was given a computer account that they were to use during the study. The shell on these accounts was instrumented to log all commands. In addition, a modified CO_2P_3S interface logged all user activity. The subjects were informed that their actions were being monitored, but were not told what was being measured.

Because of problems monitoring the subjects, the results for the study are derived from metrics that can be measured from the application code for the two programs. From the study, we expected the following:

- CO_2P_3S programmers would write less code than non–CO_2P_3S Java programmers.
- CO_2P_3S users would write less complex application code than non–CO_2P_3S Java programmers.

6.2.2 Threats to Internal Validity

Internal validity is the degree to which the data collected during a study are the result of the intended experimental factors, and not the result of other outside factors. Consider an experiment comparing user productivity in two different programming environments. If both groups of users use the same operating system, the only difference will be the programming environment so the study will be more internally valid. If the two groups used different

[2]The problem description (in Appendix D) was rewritten as a reaction and diffusion of mould and bacteria over a decomposing donut, but the problem was identical.

operating systems, then some of the observations may be the result of this difference instead, making the study less internally valid.

The threats to internal validity in this study are:

- The subjects knew that they were being monitored and so they could change their normal work habits.

- The new accounts for the study were created on the same environment as the undergraduate accounts that the subjects used on an everyday basis. Subjects could inadvertently use their normal accounts, which were not instrumented, to work on their assignments. The result would be that we would have incomplete data on the development activities of these subjects, which would limit the conclusions we could draw from the study. Note that this threat only applies to subjects writing non–CO_2P_3S Java programs, as the JVM and compiler were available to the subjects in their undergraduate accounts. While these subjects would not be able to access the supplied barrier code, this problem would not become apparent until the subjects attempted to compile their (already developed) programs and found they could not import the needed classes. In contrast, CO_2P_3S subjects could only run the system when logged on with their study accounts. This means that any underestimation in measurement for this factor would weaken the case for CO_2P_3S.

- The subjects could change the shell in their study accounts. Only the one shell was instrumented to log the activities of the subjects, so any change would prevent us from monitoring development activities. The subjects were instructed not to change the shell. Again, this error would weaken the case for CO_2P_3S.

- When developing non–CO_2P_3S Java programs, it was possible for the subjects to access and use the parallel structural code generated by CO_2P_3S rather than writing it themselves (particularly if the subjects used CO_2P_3S for first part of the study). Although code reuse is usually laudable, it does mean that our measurements will not accurately reflect program development with non–CO_2P_3S Java. Even if subjects do not reuse code, they may be influenced by the design of the framework. This situation would also weaken the case for CO_2P_3S. This is an example of a *confounding variable* in experiment design. The results of the second study may be influenced by the first study.

- In all studies, there is the possibility of *mortality*, which refers to the loss of participants in a study. In this case, it refers to subjects who do not complete the assignments. This situation would not favour non–CO_2P_3S Java or CO_2P_3S.

Note that none of the threats to internal validity promote CO_2P_3S. Therefore this is a conservative study with respect to internal validity.

6.2.3 Threats to External Validity

External validity is the degree to which the results of a study are generalizable and can be used to predict the outcome for a different group of subjects. For example, a study with too few participants may not be externally valid as it may not accurately predict the results for larger populations.

The threats to external validity in this study are:

- The size of the study is too small. There are not enough sample points to be certain that any conclusions drawn from the study can be generalized.
- The two problems for the study require only a single design pattern. Larger problems will likely require multiple CO_2P_3S pattern templates. There may be additional difficulties in composing pattern templates (and the generated frameworks) that will affect the usability of CO_2P_3S.
- The subjects were told which pattern to use in their solutions to the problems in the study. This removes a critical part of the creation of a program in a pattern–based system: selecting the most appropriate pattern. It is not clear what the impact of a pattern–based tool will have on the design process.
- Our usability study only considers novice parallel programmers. It is not clear how these results will generalize to more advanced users [99].

6.3 Results of the Usability Study

The results of our usability study are summarized in Tables 6.1 and 6.2. Note that the same set of subjects participated in both studies but their roles were reversed (non–CO_2P_3S Java students in the first study became CO_2P_3S students in the second and *vice versa*). The CO_2P_3S group wrote fewer lines of code for both applications, 40% less for the LaPlace solver and 53% less for the reaction–diffusion program. The CO_2P_3S subjects also used fewer classes, 40% fewer for LaPlace and 30% less for reaction–diffusion. The difference is the structural code generated for the Mesh pattern template in CO_2P_3S. In contrast, the non–CO_2P_3S Java solutions had to write this structure themselves. Not only is this code difficult to write, it is a considerable portion of the complete application for these two problems. It is important to mention that we did not check the submissions for correctness, so we expect that some of the submitted programs contain errors.

However, the fact that CO_2P_3S users wrote less code does not necessarily translate to a reduction in overall development effort. A small amount of complex code can be more time consuming to develop than a large amount of straightforward code. For example, recursive programs can be quite small, but some programmers (particularly novices) find them difficult to write and understand. Thus, a measure of the complexity of the application code is

Table 6.1: Code measurements from the first part of usability study, for the LaPlace Solver.

CO_2P_3S students				Non–CO_2P_3S Java students			
No. of Programs	Avg. Lines of Code	Avg. No. of Classes	Avg. No. Choice Points	No. of Programs	Avg. Lines of Code	Avg. No. of Classes	Avg. No. Choice Points
10	171.6	4.1	20.0	8	274.9	6.6	52.6

Table 6.2: Code measurements from the first part of usability study, for the reaction–diffusion problem. The set of subjects is the same as in Table 6.1 but their roles are reversed.

CO_2P_3S students				Non–CO_2P_3S Java students			
No. of Programs	Avg. Lines of Code	Avg. No. of Classes	Avg. No. Choice Points	No. of Programs	Avg. Lines of Code	Avg. No. of Classes	Avg. No. Choice Points
6	131.0	4.2	11.8	6	278.5	6.0	46.5

needed to obtain a more complete picture of the amount of effort required to write a program in CO_2P_3S.

One such complexity measure is *choice points*. Choice points are places in a program where the flow of control can be altered and may no longer be sequential. Examples of choice points are selection control statement (such as `if` and `switch` statements) and loops (which contain conditions to decide what code is executed next). Other examples of choice points can include boolean operators or exception handling. A complete list of the choice points used in this dissertation is given in Appendix G. The philosophy behind this complexity measure is that errors in programs tend to occur when the wrong code is executed after a decision point. In contrast, a program that is a simple sequence of operations will usually contain fewer errors since there are less decisions that can go wrong.

Table 6.1 shows that the CO_2P_3S programs in this study contained fewer choice points than non–CO_2P_3S Java solutions, 62% less on the LaPlace solver and 75% less in the reaction–diffusion problem. The structural code generated for the Mesh template contains most of the choice points in these applications. Recall that the application–specific code for the Mesh framework consists of methods implementing operations at the level of an individual mesh element. The iteration required for controlling the computation for the two–dimensional set of mesh elements is contained in the generated framework code, which accounts for many of the choice points in a mesh computation.

6.4 Problems with the Study

The results for the study are based on static measurements from the application code submitted by the subjects. There are other interesting aspects of usability that we would have liked to measure, such as the number of compilations, the number of program executions, and the amount of time users spent working on the problems. These measurements could have provided quantitative data on the amount of effort needed to create a parallel program in CO_2P_3S and non–CO_2P_3S Java. Unfortunately, several of the threats to internal validity described in the previous section were realized, so no conclusions could be drawn from the instrumentation. In addition, other factors that interfered with this study are discussed.

Problems with the login accounts was the principal reason that we were unable to monitor the development activities of the participants. In particular, these accounts were created in the same environment in which the subjects normally work. As a result, many subjects used their normal accounts out of habit, only switching to their study accounts when they discovered they were missing facilities needed to complete an assignment. This problem affected the non–CO_2P_3S Java group, who could have a completely developed program before they noticed any missing facilities (namely the barrier implementation). The CO_2P_3S group did not suffer from this problem since the user interface could only be started from the study accounts.

In the results from Table 6.1, the rate of mortality increased significantly for the second part of the study. This was the result of errors in the assignment specification. In particular, one problem with the reaction–diffusion specification was only uncovered the day before the assignment was due. Coupled with the fact that the assignment was only worth 3% of their final grade, some students gave up in frustration.

There were two other, smaller problems affecting only individual participants. First, while none of the subjects changed the shell for their study accounts, one did start another shell on the command line and bypassed our instrumentation. Given the more general problems with the study accounts, this problem did not affect the results. Second, another subject submitted a non–CO_2P_3S Java program that was influenced by the Mesh framework generated by CO_2P_3S. Some of the design and structural code mirrored the framework code. However, it was clear that the subject had tailored the code to the particular problem.

More generally, the environment prepared for the study was too uncontrolled. The subjects could accidently leave or avoid the instrumented environment without being aware that they had done so. Since we could not observe the subjects throughout the study, there was no way to correct this situation before it became a problem. Future studies must either have a more controlled environment or must monitor the participants more closely to ensure similar problems do not occur.

Although it was not a threat to the validity of the study, a further factor that we did not consider was how the working habits of students has changed since the usability studies of Enterprise. High speed networks to the home, such as cable modems and DSL service, have had a dramatic impact on the way students complete their assignments. In the past, students had no choice but to work in laboratories on campus. Today, many students take advantage of their Internet access to work from home in one of two ways. First, they can log into computers on campus from home. Since X Window servers can be found for all major operating systems, students can remotely display applications running on university computers on their home machines. The second way students can work from home is to download the software they need and install it locally. The available bandwidth of high–speed Internet access means that downloading large software packages is no longer an obstacle. In contrast, when the Enterprise studies were undertaken, home networking was limited to 9600 baud modems, so students had to work on campus. Students today expect a larger degree of freedom in accessing resources and software that they can use to complete course assignments.

Unfortunately, it was not possible to give the students this flexibility when participating in our study, especially when they were using CO_2P_3S. For technical reasons, the CO_2P_3S interface cannot be remotely displayed over a network.[3] Instead, the interface must be run on the local machine. In addition, the instrumentation we put into the interface to monitor user activity required access to file systems on campus, so we could not make it available to the students for download. The non–CO_2P_3S Java subjects had more flexibility; while they were asked to work in their study accounts, they could do so by logging in from home.

6.5 Issues for Future Studies

This study considered the usability of the Patterns Layer of CO_2P_3S in the hands of a group of novice parallel programmers. This is only one layer of the PDP process; the other layers also need to be assessed. As well, the environment for our study failed to provide reliable data even for our limited study. This section discusses issues that should be considered by future studiess.

6.5.1 Study Environment

Conducting usability studies for parallel programming systems as course assignments seems like a good idea. Most students have little or no experience in the subject, but are willing to learn and use new tools. Assignments are usually done over the course of about two weeks, allowing the study to use larger problems that are more comparable to real–world software development.

[3] This appears to be a limitation in the Swing graphical user interface components.

Unfortunately, the environment for this study failed to provide us with the usability data we wanted to measure for subjects writing programs in non–CO_2P_3S Java. This failure was the result of three factors:

1. The study used the same laboratory that the students used normally. Out of habit, many subjects used their existing accounts rather than the instrumented accounts created for them for the study.

2. The laboratory had a Java installation that students could use from their normal accounts.

3. Over the course of a two week assignment, it was not possible to monitor students as they worked in the laboratory.

These factors combined to allow students to develop their Java programs in an uninstrumented environment, preventing us from collecting any data on their development efforts. In addition, the students were concentrating on completing their assignments, not on their participation in the study, so it is understandable that they would forget to use the environment created for the study when it was not necessary.

However, the Enterprise studies were more successful at gathering usability information during student assignments. The difference is that the software for those studies was not widely available outside of the environment created for the study. The participants had to use the instrumented environment to complete their assignments. This added an additional measure of control over the participants in the study that we were unable to apply because of the ubiquity of Java installations.

Future studies need to concentrate on improving control over their environment so that more data can be collected. One way of providing better control is to use volunteer of paid subjects. These programmers will be more aware that they are participating in a study and will be less likely to circumvent any instrumentation. Even so, subjects should work where they can be observed to ensure they do not accidently leave the setup for the study.

The drawback to this new environment is that it limits the size of the problem that can be used. It will be difficult to find subjects willing to spend more than one or two days on a study. Larger problems, such as the reaction–diffusion example, take time to fully understand and implement properly. This time can be reduced by providing library code for the subjects to use.

6.5.2 Usability at Different Program Development Stages

The reported usability studies have a common theme: they measure the effort needed to develop a complete, working parallel application by a group of novice programmers. This is an important measure of the usability of a parallel programming system. However, it fails to address two important considerations.

First, these studies fail to consider the needs of more experienced parallel programmers. It is not clear if usability results from novice users will generalize to experts. Novice programmers will be most concerned with the abstractions for expressing parallelism and synchronization provided in the programming model of the tool. They will spend most of their time just trying to create a working program. Expert programmers have different expectations from their tools. Since they already have experience with a variety of parallel programming models, experts are usually more concerned with performance and flexibility. They expect a system that will allow them to implement the parallel structure that they feel is best for their program and to tune that structure for optimal performance. Expert programmers can become frustrated with high–level tools that do not give them control over their application code [96].

Putting this into perspective with the abstractions in the PDP process, novice users will spend most of their time working in the Patterns Layer, the highest layer of abstraction. The high–level abstractions at the Patterns Layer are invaluable to these users since they are not familiar with the difficulties of low–level parallel programming. In contrast, experts will spend more time at the Intermediate Code and Native Code layers trying to get more performance out of their programs.

The second consideration that these studies fail to consider is differences in usability at different stages of application development. Abstractions that support one stage of development may hinder efforts at other stages. For example, automatic data replication policies supports rapid application development, but hampers performance tuning if the chosen policy cannot be overridden when the choice results in poor performance.

The importance of this consideration is highlighted by the layered model in the PDP process, in which different abstractions are provided for each stage of program development. Studies for tools supporting the process, including CO_2P_3S, should assess the abstractions for each layer.

Another consideration is that previous usability studies considered program development as one large process. Instead, it consists of several smaller processes, such as the learning phase, initial development of the program, and performance tuning. Usability measurements should consider each of these phases separately [99].

6.5.3 Measuring the Learning Curve

An important usability aspect that has not been adequately addressed is measuring the learning curve for a parallel programming system. The usual benchmark for the learning curve is the time it takes for users to write their first working parallel program. Systems that require substantial time and effort to write even a simple program will probably not succeed.

An interesting question is how prior parallel programming experience influences the learning curve of a user [99]. Is it easier to use a high–level tool such as CO_2P_3S after writing parallel programs in non–CO_2P_3S Java? Does access to the parallel structural code generated by CO_2P_3S help users write parallel programs in non–CO_2P_3S Java, by providing a good example to start from? The answers to these questions can lead to better methods of teaching parallel programming.

6.6 Summary

This chapter examined the design and results of a study to assess the usability of the CO_2P_3S parallel programming system. This study was conducted over three programming assignments in an undergraduate class on the implementation of object–oriented programming languages. Because of problems in the setup, described in detail, the results are derived from static measurements that were taken from the code submitted for each of the assignments. Suggestions for future studies, that both avoid the errors in this study and cover different usability concerns, were also presented.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

This dissertation described the PDP process, a new process for developing parallel applications based on three techniques from sequential object–oriented programming: design patterns, frameworks, and multiple programming layers. This new process improves upon existing work by emphasizing *correctness* and *openness* concerns. It is intended as a basis for the next generation of pattern–based parallel programming systems, which will provide a flexible, open, and extendible environment for writing high–performance parallel applications. The CO_2P_3S parallel programming system provided a concrete example of a tool implementing the process. In addition, CO_2P_3S demonstrated additional tool support for programming with frameworks.

In more detail, the contributions of this thesis are:

- A survey of the field of parallel programming systems (Chapter 2). These systems were compared and contrasted using the 13 characteristics of ideal template–based (pattern–based) parallel programming systems. The survey also included related research on design patterns, object–oriented frameworks, and object–oriented modeling languages.

- A complete description of the PDP process for developing pattern–based parallel programs (Chapter 3). The process supports three layers of abstraction. The topmost layer, the Patterns Layer, supports pattern–based program development through framework code generation. The code generation is guided by a user–supplied description of a pattern structure using a *design pattern template*. The template allows the user to refine the pattern structure using *pattern template parameters*. This layer emphasizes correctness by generating correct structural code for the pattern and encapsulating it in an object–oriented framework to prevent the user from introducing errors. The Intermediate Code Layer provides a high–level, explicitly parallel object–oriented programming language. This layer emphasizes *openness* by providing access to the code generated at the previous layer. The last layer, the Native Code Layer, provides access to the implementation of the abstractions from the higher layers. These abstractions can be tuned for the execution environment.

- A discussion of issues that will be faced by tool developers when creating new programming systems based on the PDP process (Chapter 3).

- A description of CO_2P_3S, one implementation of the PDP process (Chapters 3 and 4). As well as supporting the process, CO_2P_3S includes additional tool and code support for using the frameworks generated from the design pattern templates. CO_2P_3S was evaluated using the 13 characteristics of ideal pattern–based parallel programming systems (Chapter 4). This evaluation showed that CO_2P_3S advances the state of the art in parallel programming systems by considering *openness* and *correctness* without sacrificing advances in other characteristics.

- A description of the design pattern templates currently supported by CO_2P_3S (Chapter 4): the Two–Dimensional Mesh, the Distributor, and the Phases. A proposed fourth template, the Pipeline, was also described. The Pipeline is unique in that is addresses the load balancing problems that plague traditional pipelines.

- A demonstration that the pattern templates can be used to create working parallel programs that provide performance gains. The Mesh was used to implement a reaction–diffusion texture generator (Chapter 3). The Distributor was used to write a parallel IDA* search program and a parallel sorting algorithm (in conjunction with the Phases pattern template) (Chapter 5). A prototype framework for the Pipeline was used to create a parallel JPEG encoder (Chapter 5).

- The results of a usability study on the CO_2P_3S system in the hands of novice users (Chapter 6). This study showed that CO_2P_3S users write fewer lines of code and that this code is less complex than an equivalent solution in non–CO_2P_3S Java using threads. Unfortunately, several problems in the study limited the amount of data that could be reliably obtained. These problems, and suggestions for ways to avoid them in the future, were also presented.

7.2 Future Work

The CO_2P_3S parallel programming system is intended as more than a concrete implementation of the PDP process. It is also intended as the basis for new research in pattern–based parallel programming systems. This section examines some of the research that remains to be done, and some of the work that is already underway.

7.2.1 New Patterns

The most obvious direction for continued research is to find more design patterns to add to CO_2P_3S as templates. The only way to find new patterns is to write parallel applications and abstract out the common structural elements (also called *pattern mining*). This work will improve the *utility* of CO_2P_3S by increasing the range of possible applications that can be created with the system.

Currently, work is underway to add a Flow design pattern template, which will add *wavefront* or *systolic* computations to CO_2P_3S. This template efficiently parallelizes the FastLSA algorithm for aligning DNA or protein sequences [23].

7.2.2 Adding and Changing Design Pattern Templates and Frameworks

A recurring problem in pattern–based systems is that users are limited to the set of patterns provided by the tool. If the system does not support the best pattern for a given application, then the user will either have to settle for a suboptimal solution or implement the structure manually.

To address this weakness, Steven Bromling has created MetaCO_2P_3S. This tool allows users to create new pattern templates and frameworks that integrate seamlessly with CO_2P_3S [18, 19]. These new templates can be shared with other CO_2P_3S users, who can import them into the user interface and use them in applications. In fact, the existing templates (the Two–Dimensional Mesh, Distributor, and Phases) have been rewritten using MetaCO_2P_3S. In contrast, the patterns supplied with DPnDP were special cases and cannot be reproduced using its framework for adding new patterns.

When MetaCO_2P_3S is released to the general user community, programmers will have an extendible parallel programming system that is not limited to the templates supplied by its designers. The user community will be able to contribute their own experiences in parallel programming to CO_2P_3S, adding new templates and modifying existing ones. The next logical step is to create a mechanism that fosters collaboration and allows users to exchange their work. One possible mechanism is a Web–based pattern repository. CO_2P_3S could include an interface to this repository so users could query for relevant templates and download the ones they wish to use in their application. They could also upload new or revised templates for use by other programmers. This kind of collaboration will improve the *utility* of CO_2P_3S by providing a much larger selection of pattern templates. Further, a user community can create templates at a much faster pace than we could by ourselves.

7.2.3 Support Tools

In addition to MetaCO_2P_3S, the CO_2P_3S system would benefit from the addition of other support tools. The tools created for Enterprise, such as the distributed debugger [50] and performance monitoring toolset [114], provide an excellent starting point for this work. However, the tools for a programming system should provide similar abstractions to those supported by the programming model, so users do not need to learn new concepts to use them. Since the PDP process supports several layers of abstraction, it follows that the support tools should also support different abstractions. Specifically, a support tool should not expose any features of a lower layer.

Another possible support tool would be a *pattern language system* for designing parallel programs. CO_2P_3S, like most parallel programming systems, has no support for the design phase of application development. The system assumes that the user already has a design and concentrates on supporting the implementation phase. A design tool would greatly improve the usability of CO_2P_3S in the hands of novice parallel programmers. Also, as the number of pattern templates grows (possibly through the repository discussed in the previous section), they must be organized so that users can easily find the best template. A design tool based on a pattern language can provide this organization.

7.2.4 Supporting Different Architectures

Currently, CO_2P_3S generates multi–threaded Java framework code that is targeted at shared–memory multiprocessors. Work is underway to create a distributed–memory version of the CO_2P_3S frameworks for networks of workstations using Java Remote Method Invocation and Jini. Ideally, this port will only affect the structural framework code, and have no effect on the set of pattern templates or the set of hook methods. However, there are performance issues that must be addressed.

7.2.5 Language Issues

The work presented in this dissertation concentrated on the Patterns Layer. The Intermediate Code and Native Code layers are still active research areas. Some of the problems that have yet to be resolved are:

- What high–level primitives should be included at the Intermediate Code Layer?
- How should these be translated to the Native Code Layer?
- How do we permit users to introduce new high–level primitives to the Intermediate Code Layer?

The last problem is the most interesting. In keeping with the spirit of openness in CO_2P_3S, advanced users should be free to customize and extend the system to meet their needs at all of the layers supported by the PDP process, not just the Patterns Layer.

Another language issue to consider is the addition of features at the Intermediate Code Layer to support frameworks, in addition to supporting parallelism and synchronization. The language changes introduced by CORRELATE (Section 2.3.3) to simplify framework instantiation are an example. Other language features, such as multi–method dispatch [30], adding source code to interfaces, and aspects [57] could also be included. These features may make it easier to modify or augment the generated framework code, enhancing the usability of the lower layers of CO_2P_3S.

7.2.6 Preserving Low–level Changes During High–level Code Regeneration

The implicit assumption in the layered programming model in the PDP process is that the user starts at the Patterns Layer and always works downward, to the Intermediate Code Layer and finally to the Native Code Layer. We have only considered one scenario where the user moves upward through the layers. In this scenario, if a user introduces errors into the structure of an application while tuning it at a lower layer, then the correct structure can be regenerated at the Patterns Layer from the pattern template (Section 3.2.4).

However, it is possible to envision a scenario where a user may need to modify some part of a pattern template and regenerate some of the structural code while preserving optimizations made at lower layers. For example, a user may generate a Mesh framework and optimize it by adding a parallel reduction to gather the results of the computation. Later, the user may discover that they selected the wrong topology for the problem, and would like to change it without losing the implementation of the reduction when the framework is regenerated.

One possible solution to this problem is use MetaCO_2P_3S to add the optimization as a parameter in the Mesh pattern template. The code for the parallel reduction would then be generated at the Patterns Layer, where the topology can be changed easily. However, this solution is best with commonly recurring optimizations. Otherwise, the pattern template can become cluttered with many parameters for options that are rarely used, making it more difficult to use.

For those cases when the user does not change the pattern template, there should be a mechanism for partially regenerating the structural code. This mechanism should separate the different aspects of a framework and allow a user to specify which parts of the structure should be replaced when regenerating code for a pattern template.

7.2.7 Usability Studies

The usability study from Chapter 6 provides some data on the benefits of CO_2P_3S and, by association, the benefits of the PDP process. Clearly, more thorough usability studies are needed. In particular, an experiment that avoids the problems from Section 6.4 should be undertaken. The suggested improvements from Section 6.5 will also provide better experiments that should reveal more interesting aspects of pattern–based parallel programming with CO_2P_3S.

7.3 Conclusion

Parallel architectures provide the opportunity for large performance benefits to programmers who can take advantage of them. Unfortunately, harnessing this processing power is difficult. Parallel programs are harder to write and debug than sequential programs. They must include communication and synchronization code, which is difficult to implement correctly. They may be non–deterministic, which can hamper debugging efforts. Finally, they should be tuned for efficiency, which requires knowledge of the parallel architecture.

The best way to contend with the added complexity of parallel programming is by providing tools, such as parallel programming systems and languages. These tools can provide abstractions that address some of this complexity, simplifying the task of creating parallel applications.

The PDP process provides a basis for a new generation of pattern–based parallel programming systems based on design patterns, object–oriented frameworks, and multiple programming layers. Tools supporting this new process will provide an open, flexible environment for writing high–performance parallel programs. The CO_2P_3S parallel programming system demonstrates the feasibility of this process, and will also serve as the basis for future research in this area.

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object–Oriented Concurrent Computing*, pages 37–53. MIT Press, 1987.

[3] S. Alpert and R. Lam. The ultimately publishable computer science paper for the latter '90s: A tip for authors. *Communications of the ACM*, 40(1):94, 1997.

[4] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 30th Conference of the American Federation of Information Processing Societies*, pages 483–485, 1967.

[5] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[6] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, and G. Gao. ABC++: Concurrency by inheritance in c++. *IBM Systems Journal*, 34(1):120–136, 1995.

[7] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison–Wesley, third edition, 2000.

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

[9] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

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

[11] K. Beck and R. Johnson. Patterns generate architecture. In *Proceedings of the 8th European Conference on Object–Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 139–149. Springer–Verlag, 1994.

[12] 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, 1993.

[13] A. Beguelin 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.

[14] A. Beguelin and V. Sunderam. Tools for monitoring, debugging, and programming in PVM. In *Proceedings of the Third European PVM Conference (EuroPVM'96)*, volume 1156 of *Lecture Notes in Computer Science*, pages 7–13. Springer-Verlag, 1996.

[15] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison–Wesley, 1999.

[16] J. Bosch. Design pattern and frameworks: On the issue of language support. In *Object–Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357 of *Lecture Notes in Computer Science*, pages 133–136. Springer–Verlag, 1997.

[17] M. Brockington and J. Schaeffer. APHID: Asynchronous parallel game–tree search. *Journal of Parallel and Distributed Computing*, 60(2):247–273, 2000.

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

[19] S. Bromling, D. Szafron, J. Schaeffer, S. MacDonald, and J. Anvik. Generalising pattern–based parallel programming systems. *Parallel Computing*, 2001. To appear.

[20] R. Bruce, S. Chapple, N. MacDonald, A. Trew, and S. Trewin. CHIMP and PUL: Support for portable parallel computing. In *Proceedings of the Fourth Annual Conference of the Meiko User Society*, 1993. Also available at http://www.epcc.ed.ac.uk/epcc-projects/PUL/dox.html.

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

[22] K. Mani Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, 1992.

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

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

[25] R. Crawford. Usenet posting in comp.parallel, 1999.

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

[27] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[28] J. Dongarra, S. Browne, and K. London. Review of performance analysis tools for MPI parallel programs. *NHSE Review*, 3(1), 1998. http://www.nhse.org/NHSEreview.

[29] D. D'Souza and A. Wills. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison–Wesley, 1998.

[30] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi–dispatch in the java virtual machine: Design and implementation. In *Proceedings of 6th Usenix Conference on Object–Oriented Technologies and Systems (COOTS'2001)*, pages 77–92, 2001.

[31] 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, 1993.

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

[33] I. Foster. *Designing and Building Parallel Programs*. Addison–Wesley, 1995.

[34] M. Frigo, C. Leiserson, and K. Randall. The implementation of the cilk–5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*.

[35] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into object–oriented application frameworks. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 491–502, 1997.

[36] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Reusing hooks. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, chapter 9, pages 219–236. Wiley, 1999.

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

[38] A Geist, A Beguelin, J Dongarra, W Jiang, R Manchek, and V Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[39] D. Goswami, A. Singh, and B. Priess. Architectural skeletons: The reusable building-blocks for parallel applications. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applciations (PDPTA'99)*, pages 1250–1256, 1999.

[40] D. Goswami, A. Singh, and B. Priess. Using object–oriented techniques for realizing parallel architectural skeletons. In *Proceedings of the Third International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'99)*, volume 1732 of *Lecture Notes in Computer Science*, pages 130–141. Springer–Verlag, 1999.

[41] D. Goswami, A. Singh, and B. Priess. Building parallel applications using design patterns. In H. Erdogmus and O. Tanir, editors, *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*. Springer–Verlag, 2000.

[42] A. Grimshaw. Easy to use object–oriented parallel programming with mentat. *IEEE Computer*, 26(5):39–51, 1993.

[43] M. Gupta, J. Choi, and M. Hind. Optimizing java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object–Oriented Programming (ECOOP 2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 422–446. Springer–Verlag, 2000.

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

[45] E. Hamilton. *JPEG File Interchange Format, Version 1.02*, 1992. Available at http://www.w3.org/Graphics/JPEG.

[46] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML designs to java. *ACM SIGPLAN Notices*, 35(10):178–187, 2000. Proceedings of 2000 Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA 2000).

[47] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[48] S. Ben Hassen, H. Bal, and C. Jacobs. A task and data parallel programming language based on shared objects. *ACM Transactions on Programming Languages and Systems*, 20(6):1131–1170, 1998.

[49] W. Hui, S. MacDonald, J. Schaeffer, and D. Szafron. Visualizing object and method granularity for program parallelization. In *Proceedings of the Twelfth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2000)*, pages 286–291, 2000.

[50] P. Iglinski. An execution replay facility and event–based debugger for the enterprise parallel programming system. Master's thesis, Department of Computing Science, University of Alberta, 1994.

[51] ISO/IEC JTC1 SC29 Working Group 1 (Joint Photographic Experts Group). *Digital Compression and Coding of Continuous–Tone Still Images, Part 1: Requirements and Guidelines, ISO/IEC International Standard 10918–1 (Also ITU–T T.81)*, 1992.

[52] ISO/IEC JTC1 SC29 Working Group 1 (Joint Photographic Experts Group). *Digital Compression and Coding of Continuous–Tone Still Images, Part 3: Extensions, ISO/IEC International Standard 10918–3 (Also ITU–T T.84)*, 1997.

[53] R. Johnson. Documenting frameworks using patterns. *ACM SIGPLAN Notices*, 27(10):63–76, 1992. *Proceedings of the 1992 Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA'92).*

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

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

[56] W. Karpoff and B. Lake. PARDO–a deterministic, scalable programming paradigm for distributed memory parallel computer systems and workstation clusters. In *Proceedings of Supercomputing Symposium*, pages 145–152, 1993.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the Fifteenth European Conference on Object–Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Spring–Verlag, 2001.

[58] R. Korf. Depth–first iterative–deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[59] R. Lavender and D. Schmidt. Active object: An object behavioral pattern for concurrent programming. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*, chapter 30, pages 483–499. Addison–Wesley, 1996.

[60] D. Lea. Six misconceptions about reliable distributed computing. Distributed Objects Mailing List Archive, http://www.distributedcoalition.org/mailing_lists/dist-obj/msg01163.html, 1998.

[61] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley, second edition, 1999.

[62] D. Lea. *Overview of package `util.concurrent` Release 1.2.6*, 1999. Available at http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/-util/concurrent/intro.html.

[63] D. Lea. A java fork/join framework. In *Proceedings of the 2000 ACM Java Grande Conference*, pages 36–43, 2000.

[64] G. Lobe. The enterprise user interface and program animation components. Master's thesis, Department of Computing Science, University of Alberta, 1993.

[65] S. MacDonald. *MethodThread Version 3.0*, 1998. Available at http://www.cs.ualberta.ca/∼stevem/MethodThread/index.html.

[66] S. MacDonald, J. Schaeffer, and D. Szafron. Pattern–based object–oriented parallel programming. In *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 267–274. Springer–Verlag, 1997.

[67] S. MacDonald, D. Szafron, and J. Schaeffer. Object–oriented pattern–based parallel programming with automatically generated frameworks. In *Proceedings of the 5th USENIX Conference on Object–Oriented Technology and Systems (COOTS'99)*, pages 29–43, 1999.

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

[69] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. From patterns to frameworks to parallel programs. *Journal of Parallel and Distributed Computing*, 2001. Under submission.

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

[71] B. Massingill. Experiments with program parallelization using archetypes. In *Parallel and Distributed Processing: Workshops held in conjunction with the 12th International Parallel Programming Symposium and the 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*, volume 1388 of *Lecture Notes in Computer Science*, pages 844–856. Springer–Verlag, 1998.

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

[73] 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, 1993.

[74] F. Matthijs, W. Joosen, B. Robben, B. Vanhaute, and P. Verbaeten. Multi–level patterns. In *Object–Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357 of *Lecture Notes in Computer Science*, pages 112–115. Springer–Verlag, 1998.

[75] M. Mattsson and J. Bosch. Framework composition: Problems, causes, and solutions. In *Proceedings of the Twenty–Third International Conference on Technology of Object–Oriented Languages and Systems (TOOLS USA'97)*, pages 203–214, 1997.

[76] M. Mattsson and J. Bosch. Composition problems, causes, and solutions. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, chapter 20, pages 467–487. Wiley & Sons, 1999.

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

[78] P. Newton and J. Dongarra. Overview of VPE: A visual environment for message–passing. In *Proceedings of the Fourth Heterogeneous Computing Workshop*, pages 85–92, 1995.

[79] ObjectSpace, Inc. *ObjectSpace JGL: The Generic Collection Library for Java Version 3.0*, 1997. http://www.objectspace.com.

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

[81] J.-L. Pazat. Tools for high performance FORTRAN: A survey. In *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 134–158. Springer–Verlag, 1996.

[82] M. Philippsen. Data parallelism in java. In *Proceedings of the 12th Annual Symposium on High Performance Computing Systems (HPCS'98)*, pages 85–99. Kluwer Academic Press, 1998.

[83] M. Philippsen and M. Zenger. Javaparty – transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[84] L. Prechelt and B. Unger. A series of controlled experiments on design patterns: Methodology and results. *Softwaretechnik–Trends*, 18(3):53–60, 1998.

[85] L. Prechelt, B. Unger, W. Tichy, P. Brössler, and L. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 2000. To appear.

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

[87] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorgbe. The architecture of a UML virtual machine. In *Proceedings of the 2001 Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA 2001)*, pages 327–341, 2001.

[88] D. Roberts and R. Johnson. Patterns for evolving frameworks. In R. Martin, D. Riehle, F. Buschmann, and J. Vlissides, editors, *Pattern Languages of Program Design*, volume 3, chapter 26, pages 471–486. Addison–Wesley, 1998.

[89] 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.

[90] D. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *Proceedings of the 12th Sun Users Group Conference*, 1994. A list of the individual patterns can be found at http://www.cs.wustl.edu/~schmidt/patterns-ace.html.

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

[92] M. Sefika, A. Sane, and R. 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, 1996.

[93] B. Selic, G. Gullekson, and P. Ward. *Real–Time Object–Oriented Modeling*. Wiley & Sons, 1994.

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

[95] 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, 1991.

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

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

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

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

[100] L. Tahvildari. Assessing the impact of using design–patterns–based systems. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, 1999.

[101] L. Tahvildari and A. Singh. Impact of using pattern–based systems on the qualities of parallel applications. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, pages 1713–1719, 2000.

[102] The C3 Team. Chrysler goes to "extremes". *Distributed Computing*, pages 24–28, October 1998.

[103] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API)*, 1996. ANSI/IEEE Std. 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C language].

[104] S. Trewin. PUL–SM prototype user guide. Technical Report EPCC–KTP–PUL–SM–PROT–UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1993.

[105] S. Trewin, R. Baxter, and R. Davey. PUL–MD prototype user guide. Technical Report EPCC–KTG–PUL–MD–PROT–UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1996.

[106] K. van Reeuwijk, A. van Gemund, and H. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.

[107] A. Vermeulen, G. Beged-Dov, and P. Thompson. The pipeline design pattern. In *Proceedings of OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*, 1995. http://www.cs.wustl.edu/∼schmidt/OOPSLA-95/index.html.

[108] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison–Wesley, 1998.

[109] E. West and A. Grimshaw. Braid: Integrating task and data parallelism. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, pages 211–219, 1995.

[110] G. Wilson. Assessing the usability of parallel programming systems: The cowichan problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, 1994.

[111] G. Wilson. High performance programming for computational scientists. In *High Performance Computing Systems and Applications (Proceedings of the 13th Annual Symposium on High Performance Computing Systems (HPCS'99))*, volume 541 of *The Kluwer International Series in Engineering and Computer Science*, pages 7–14, 2000.

[112] G. Wilson and H. Bal. Using the cowichan problems to assess the usability of orca. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.

[113] A. Witkin and M. Kass. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):299–308, 1991.

[114] D. Woloschuk. Enterprise performance monitoring tools. Master's thesis, Department of Computing Science, University of Alberta, 1996.

[115] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 93–103, 1992.

[116] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, 1998.

[117] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura. Implementing concurrent object–oriented languages on multicomputers. *IEEE Parallel and Distributed Technology*, 1(2):49–61, 1993.

[118] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object–oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object–Oriented Concurrent Computing*, pages 55–89. MIT Press, 1987.

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

Appendix A

Source Code for the Reaction–Diffusion Example Program

This appendix contains all of the user code for the reaction–diffusion example program. It does not include the code generated for the Mesh framework. The `MorphogenPair` class is generated with the rest of the framework. The remaining classes are not part of the framework, but rather are additional classes taken from the sequential implementation of this problem.

A.1 MorphogenPair.java

This class provides all of the hook methods for the Mesh framework. It is generated by CO_2P_3S but has application–specific code inserted by the user. For this application, this class represents a pair of morphogens for a given region on a two–dimensional surface.

The user–supplied application code is indicated by bars on the left. Recall that the signatures of the hook methods are generated with this class, and are not written by the user.

```
import java.lang.* ;
// Enter additional imports here.
import java.util.Random ;

public class MorphogenPair
    extends Object
{
    public MorphogenPair(int i, int j, int surfaceWidth, int surfaceHeight,
                         Object initializer)
    {
        Random gen = (Random) initializer ;

        this._morphogen1 = new Morphogen((1.0d - (gen.nextDouble() * 2.0d)),
                                         XDIFFUSION1, YDIFFUSION1) ;
```

```
        this._morphogen2 = new Morphogen((1.0d - (gen.nextDouble() * 2.0d)),
                                        XDIFFUSION2, YDIFFUSION2) ;
    } /* MorphogenPair */

    public void initialize()
    {
    } /* initialize */

    public boolean notDone()
    {
        return(!(this._morphogen1.hasConverged() &&
                this._morphogen2.hasConverged())) ;
    } /* notDone */

    public void prepare()
    {
        this._morphogen1.updateConcentration() ;
        this._morphogen2.updateConcentration() ;
    } /* prepare */

    public void interiorNode(MorphogenPair left, MorphogenPair right,
                             MorphogenPair up, MorphogenPair down)
    {
        this._morphogen1.simulate(left.getMorphogen1(),
                                  right.getMorphogen1(),
                                  up.getMorphogen1(),
                                  down.getMorphogen1(),
                                  this._morphogen2, 1) ;
        this._morphogen2.simulate(left.getMorphogen2(),
                                  right.getMorphogen2(),
                                  up.getMorphogen2(),
                                  down.getMorphogen2(),
                                  this._morphogen1, 2) ;
    } /* interiorNode */

    public void postProcess()
    {
        this._morphogen1.updateConcentration() ;
    } /* postProcess */

    public void reduce(int i, int j, int surfaceWidth, int surfaceHeight,
                       Object reducer)
    {
        double[][] array = (double[][]) reducer ;

        array[i][j] = this._morphogen1.getConcentration() ;
    } /* reduce */

    // Enter additional user code here.
    public final Morphogen getMorphogen1()
    {
        return(this._morphogen1) ;
    } /* getMorphogen1 */
```

```
    public final Morphogen getMorphogen2()
    {
        return(this._morphogen2) ;
    } /* getMorphogen2 */

    protected Morphogen _morphogen1 ;
    protected Morphogen _morphogen2 ;

    protected static final double XDIFFUSION1 = 2.0d ;
    protected static final double YDIFFUSION1 = 1.0d ;
    protected static final double XDIFFUSION2 = 2.0d ;
    protected static final double YDIFFUSION2 = 1.5d ;
} /* MorphogenPair */
```

A.2 Main.java

This class is the full mainline method for the reaction–diffusion problem. At the end of the computation, the final results may be displayed in an output window.

```
import java.lang.* ;
import java.util.Random ;

public class Main
{
    public static void main(String[] argv)
    {
        int surfaceWidth = 0 ;
        int surfaceHeight = 0 ;
        int meshWidth = 0 ;
        int meshHeight = 0 ;
        long startTime ;
        RDMesh mesh ;

        startTime = System.currentTimeMillis() ;

        if (argv.length < 2 || argv.length > 3) {
            Main.usage() ;
            System.exit(-1) ;
        } /* if */

        try {
            surfaceHeight = Integer.parseInt(argv[0]) ;
            surfaceWidth = surfaceHeight ;
            meshWidth = Integer.parseInt(argv[1]) ;
            if (argv.length == 3) {
                meshHeight = Integer.parseInt(argv[2]) ;
            } else {
                meshHeight = meshWidth ;
            } /* if */
```

```
    } catch (NumberFormatException ne) {
        System.err.println("Illegal integer argument given.") ;
        System.exit(−1) ;
    } /* try */

    // Set morphogen constants for the simulation.
    Morphogen.setMorphogenConstants(1.0d, 1.0d, 0.02d, 1.0d) ;

    Random initializer = new Random(1) ;
    double[][] reducer = new double[surfaceWidth][surfaceHeight] ;

    mesh = new RDMesh(surfaceWidth, surfaceHeight,
                      meshWidth, meshHeight, initializer, reducer) ;

    System.out.println("Starting simulation at time " +
                       (System.currentTimeMillis() − startTime) +
                       "...") ;

    mesh.launch() ;

    System.out.println("Simulation converged.") ;

    if (Main.USEWINDOW) {
        Main.createWindow(reducer, surfaceWidth, surfaceHeight) ;
        Main.getWindow().popupWindow() ;
        Main.getWindow().setSurface(reducer) ;
        Main.getWindow().redisplayContents() ;
    } /* if */
} /* main */

public static MorphogenDisplay getWindow()
{
    return(Main._window) ;
} /* getWindow */

protected static void usage()
{
    System.out.println("Main surfaceSize meshWidth <meshHeight>\n") ;
    System.out.println("    Surface height and width are set to " +
                       "surfaceSize.") ;
    System.out.println("    Mesh height defaults to meshWidth if not" +
                       " specified.") ;
} /* usage */

protected static void createWindow(double[][] surface,
                                   int surfaceWidth, int surfaceHeight)
{
    Main._window = new MorphogenDisplay("Reaction-Diffusion", surface,
                                        surfaceWidth, surfaceHeight,
                                        DISPLAY_SCALE) ;
} /* createWindow */

protected static MorphogenDisplay _window ;
protected static final boolean USEWINDOW = true ;
```

```
    protected static final int DISPLAY_SCALE = 2 ;
} /* Main */
```

A.3 Morphogen.java

This class represents a single morphogen. It is responsible for computing a new value for itself based on its current concentration, the concentration of its neighbours, and the concentration of the other morphogen.

```
import java.lang.* ;

public class Morphogen
{
    /*********************************************************************
     * Static constants for this simulation.
     *********************************************************************/

    // Dissipation, reaction, and diffusion rates are constants for this
    // program, so they are saved as static variables. Also, several
    // other "constant" terms (i.e. those that are constant given the
    // set of constants) are also precomputed and stored. So, to ensure
    // these are consistent, clients of this class must use the
    // setMorphogenConstants() method to set the value of all constants at
    // once.
    // Be careful in here; there are various constants that are also
    // included, so be sure you don't accidently include other terms.
    public static void setMorphogenConstants(double dissipation,
                                             double reaction,
                                             double timeStep,
                                             double distance)
    {
        Morphogen.setDissipation(dissipation) ;
        Morphogen.setReaction(reaction) ;
        Morphogen.setTimeStep(timeStep) ;
        Morphogen.setDistanceTerm(2.0d * distance * distance) ;
        Morphogen.setDistanceTermTimesDissipation(Morphogen.getDissipation()
                                                  * Morphogen.getDistanceTerm()) ;
    } /* setMorphogenConstants */

    public static double getDissipation()
    {
        return(Morphogen._dissipation) ;
    } /* getDissipation */

    public static double getReaction()
    {
        return(Morphogen._reaction) ;
    } /* getReaction */

    public static double getTimeStep()
    {
```

```
    return(Morphogen._timeStep) ;
} /* getTimeStep */

public static double getDistanceTerm()
{
    return(Morphogen._distTerm) ;
} /* getYDiffusionTerm */

// Another constant term used regularly.
public static double getDistanceTermTimesDissipationTerm()
{
    return(Morphogen._distTermTimesDissipation) ;
} /* getDistanceTermTimesDissipation */

protected static void setDissipation(double dissipation)
{
    Morphogen._dissipation = dissipation ;
} /* setDissipation */

protected static void setTimeStep(double timeStep)
{
    Morphogen._timeStep = timeStep ;
} /* setTimeStep */

protected static void setReaction(double reaction)
{
    Morphogen._reaction = reaction ;
} /* setReaction */

// Note that the distance term is calculated once from the distance,
// so that this term doesn't need to be calculated multiple times.
protected static void setDistanceTerm(double distanceTerm)
{
    Morphogen._distTerm = distanceTerm ;
} /* setDistanceTerm */

protected static void setDistanceTermTimesDissipation(double term)
{
    Morphogen._distTermTimesDissipation = term ;
} /* setDistnaceTimesDissipation */

protected static double _dissipation ;
protected static double _reaction ;
protected static double _timeStep ;
protected static double _distTerm ;
protected static double _distTermTimesDissipation ;

/*******************************************************************
 * Instance state and operations.
 *******************************************************************/

public Morphogen(double initialConcentration,
                double xDiffusion, double yDiffusion)
{
```

```
    this.setConcentration(initialConcentration) ;
    this.setReadConcentration(initialConcentration + 2.0d) ;
    this.setXDiffusionTerm(2.0d * xDiffusion * xDiffusion) ;
    this.setYDiffusionTerm(2.0d * yDiffusion * yDiffusion) ;
    this.setTotalDiffusion(-2.0d * (this.getXDiffusionTerm() +
                                   this.getYDiffusionTerm())) ;
} /* Morphogen */

public final void simulate(Morphogen leftMorph, Morphogen rightMorph,
                           Morphogen upMorph, Morphogen downMorph,
                           Morphogen otherChemMorph, int morphNumber)
{
    double newValue ;
    double current = this.getConcentration() ;
    double left = leftMorph.getConcentration() ;
    double right = rightMorph.getConcentration() ;
    double up = upMorph.getConcentration() ;
    double down = downMorph.getConcentration() ;
    double otherConcentration = otherChemMorph.getConcentration() ;

    double xDiff = this.getXDiffusionTerm() ;
    double yDiff = this.getYDiffusionTerm() ;

    newValue = xDiff * right ;
    newValue += xDiff * left ;
    newValue += yDiff * up ;
    newValue += yDiff * down ;

    newValue += current * (this.getTotalDiffusion() -
                      Morphogen.getDistanceTermTimesDissipationTerm()) ;

    newValue += newValue / Morphogen.getDistanceTerm() ;

    if (morphNumber == 1) {
        if (current > otherConcentration) {
            newValue += Morphogen.getReaction() ;
        } /* if */
    } else {
        if (otherConcentration > current) {
            newValue += Morphogen.getReaction() ;
        } /* if */
    } /* if */

    newValue *= Morphogen.getTimeStep() ;
    this.setConcentration(current + newValue) ;
} /* simulate */

public final boolean hasConverged()
{
    return(Math.abs(this._concentration -
                   this.getConcentration()) < THRESHOLD) ;
} /* hasConverged */

public final double getConcentration()
```

```
{
    return(this._readConcentration) ;
} /* getConcentration */

protected final void setConcentration(double value)
{
    this._concentration = value ;
} /* setConcentration */

protected final void setReadConcentration(double value)
{
    this._readConcentration = value ;
} /* setReadConcentration */

protected final double getXDiffusionTerm()
{
    return(this._xDiffTerm) ;
} /* getXDiffusionTerm */

protected final double getYDiffusionTerm()
{
    return(this._yDiffTerm) ;
} /* getYDiffusionTerm */

// Return the sum of the x and y diffusion terms.
protected final double getTotalDiffusion()
{
    return(this._totalDiffTerm) ;
} /* getTotalDiffusion */

protected void setXDiffusionTerm(double term)
{
    this._xDiffTerm = term ;
} /* setXDiffusionTerm */

protected void setYDiffusionTerm(double term)
{
    this._yDiffTerm = term ;
} /* setYDiffusionTerm */

protected void setTotalDiffusion(double term)
{
    this._totalDiffTerm = term ;
} /* setXDiffusion */

public void updateConcentration()
{
    this._readConcentration = this._concentration ;
} /* updateConcentrations */

protected double _concentration ;
protected double _xDiffTerm ;
protected double _yDiffTerm ;
protected double _totalDiffTerm ;
```

```
    protected double _readConcentration ;

    protected static final double THRESHOLD = 0.02d ;
} /* Morphogen */
```

A.4 MorphogenDisplay.java

This class translates the final concentration values to grey–scale values for the generic display window.

```
public class MorphogenDisplay
{
    public MorphogenDisplay(String title, double[][] surface,
                            int surfaceWidth, int surfaceHeight, int scale)
    {
        this.setWindow(new GreyScaleDisplayWindow(title, surfaceWidth,
                                                  surfaceHeight, 2)) ;
        this.setSurface(surface) ;
        this.setWidth(surfaceWidth) ;
        this.setHeight(surfaceHeight) ;
    } /* MorphogenDisplay */

    public void popupWindow()
    {
        this.setPixels() ;
        this.getWindow().popupWindow() ;
    } /* popupWindow */

    public void redisplayContents()
    {
        this.setPixels() ;
        this.getWindow().redisplayContents() ;
    } /* redisplayContents */

    public void setPixels()
    {
        // Use single-element arrays to simulate arguments-by-reference.
        double[] minConcentration = new double[1] ;
        double[] maxConcentration = new double[1] ;
        double base ;

        this.findMinMaxConcentration(minConcentration, maxConcentration) ;
        base = this.setBaseGreyLevel(minConcentration[0],
                                     maxConcentration[0]) ;
        this.setWindowPixels(base, minConcentration[0]) ;
    } /* setPixels */

    // Use single-element arrays to simulate arguments-by-reference.
    protected void findMinMaxConcentration(double[] minimum, double[] maximum)
    {
        int i ;
```

```
    int j ;
    double[][] surface ;
    int width ;
    int height ;
    double concentration ;
    double min ;
    double max ;

    surface = this.getSurface() ;
    width = this.getWidth() ;
    height = this.getHeight() ;

    min = surface[0][0] ;
    max = min ;
    for(i = 0;i < width;++i) {
        for(j = 0;j < height;++j) {
            concentration = surface[i][j] ;
            if (concentration > max) {
                max = concentration ;
            } else if (concentration < min) {
                min = concentration ;
            } /* if */
        } /* for */
    } /* for */
    minimum[0] = min ;
    maximum[0] = max ;
} /* findMinMaxConcentration */

protected double setBaseGreyLevel(double min, double max)
{
    double base ;

    // Set the base level (50% grey). The first condition should
    // never occur in a randomly initialized surface, but is included
    // to be safe.
    base = max − min ;
    if (base < 0.001) {
        base = 2.0d ;
    } /* if */
    return(base) ;
} /* setBaseGreyLevel */

protected void setWindowPixels(double base, double min)
{
    int i ;
    int j ;
    GreyScaleDisplayWindow window ;
    int width ;
    int height ;
    double[][] surface ;
    double nGreys ;

    surface = this.getSurface() ;
    width = this.getWidth() ;
```

```
    height = this.getHeight() ;
    window = this.getWindow() ;
    nGreys = (double) GreyScaleDisplayWindow.getNumberOfGreys() ;

    for(i = 0;i < width;++i) {
        for(j = 0;j < height;++j) {
            window.setPixel(i, j, (int)
                            (((surface[i][j] - min) * nGreys) / base)) ;
        } /* for */
    } /* for */
} /* setWindowPixels */

protected double getSurface(int i, int j)
{
    return(this._surface[i][j]) ;
} /* getSurface */

public void setSurface(int i, int j, double value)
{
    this._surface[i][j] = value ;
} /* setSurface */

public int getWidth()
{
    return(this._width) ;
} /* getWidth */

public int getHeight()
{
    return(this._height) ;
} /* getHeight */

protected double[][] getSurface()
{
    return(this._surface) ;
} /* getSurface */

public void setSurface(double[][] surface)
{
    this._surface = surface ;
} /* setSurface */

protected void setWidth(int width)
{
    this._width = width ;
} /* setWidth */

protected void setHeight(int height)
{
    this._height = height ;
} /* setHeight */

protected GreyScaleDisplayWindow getWindow()
{
```

```
        return(this._window) ;
    } /* getWindow */

    protected void setWindow(GreyScaleDisplayWindow window)
    {
        this._window = window ;
    } /* setWindow */

    protected GreyScaleDisplayWindow _window ;
    protected double[][] _surface ;
    protected int _width ;
    protected int _height ;
} /* MorphogenDisplay */
```

A.5 GreyScaleDisplayWindow.java

This class provides a generic grey–scale display window for visualizing the final result of the computation.

```
import java.lang.* ;
import java.awt.* ;

public class GreyScaleDisplayWindow
{

    // Constant for the number of greys available to display.
    public static int getNumberOfGreys()
    {
        return(NUM_GREYS) ;
    } /* getNumberOfGreys */

    protected static int NUM_GREYS = 255 ;

    public GreyScaleDisplayWindow(String title, int width, int height,
                                  int scale)
    {
        this._canvas = new DisplayCanvas(new Dimension(width, height),
                                         scale) ;
        this._frame = new FrameDisplay(title, this._canvas) ;
    } /* GreyScaleDisplayWindow */

    public void popupWindow()
    {
        this._frame.display() ;
    } /* display */

    public void redisplayContents()
    {
        Graphics g = this._canvas.getGraphics() ;
        this._canvas.paint(g) ;
        g.dispose() ;
```

```
    } /* redisplay */

    public void setPixel(int x, int y, int intensity)
    {
        this._canvas.setPixel(x, y, intensity) ;
    } /* setPixel */

    protected DisplayCanvas _canvas ;
    protected FrameDisplay _frame ;
} /* GreyScaleDisplayWindow */

class FrameDisplay extends Frame
{
    public FrameDisplay(String title, DisplayCanvas canvas)
    {
        super(title) ;
        this.setLayout(new FlowLayout()) ;
        this.add(canvas) ;
    } /* FrameDisplay */

    public void display()
    {
        this.pack() ;
        this.show() ;
    } /* display */
} /* FrameDisplay */

class DisplayCanvas extends Canvas
{
    public DisplayCanvas(Dimension size, int scale)
    {
        int i ;
        int j ;

        this.setSize(new Dimension(size.width * scale,
                                   size.height * scale)) ;
        this._scale = scale ;
        this._size = size ;

        // Make a set of flyweight colors.
        int nGreys = GreyScaleDisplayWindow.getNumberOfGreys() + 1 ;
        this._colours = new Color[nGreys] ;
        for(i = 0;i < nGreys;++i) {
            this._colours[i] = new Color(i, i, i) ;
        } /* for */

        this._cells = new Color[this._size.width][this._size.height] ;
        for(i = 0;i < this._size.width;++i) {
            for(j = 0;j < this._size.height;++j) {
                this._cells[i][j] = this._colours[nGreys - 1] ;
            } /* for */
        } /* for */
    } /* DisplayCanvas */
```

```
    public void paint(Graphics g)
    {
        int i ;
        int j ;

        for(i = 0;i < this._size.width;++i) {
            for(j = 0;j < this._size.height;++j) {
                g.setColor(this._cells[i][j]) ;
                g.fillRect(this._scale * i, this._scale * j,
                           this._scale, this._scale) ;
            } /* for */
        } /* for */
    } /* paint */

    public void setPixel(int x, int y, int intensity)
    {
        this._cells[x][y] = this._colours[intensity] ;
    } /* setPixel */

    protected int _scale ;
    protected Dimension _size ;
    protected Color[][] _cells ;

    // Make a set of flyweight colours.
    protected Color[] _colours ;
} /* DisplayCanvas */
```

Appendix B

Design Pattern Template Description Format

This appendix describes the format for documenting the design pattern templates in CO_2P_3S. It follows the traditional for design patterns from [37] at the beginning, which preserves the instructional nature of design patterns. However, this documentation format diverges to discuss the pattern template parameters supported by CO_2P_3S and to describe the framework generated at the Patterns Layer. This documentation does not include information about the frameworks for the Intermediate Code and Native Code layers. Note that the example documentation in Appendix C also includes an introduction section. The introduction is not part of normal documentation, but was used to introduce several concepts to CO_2P_3S users.

B.1 Intent

This section introduces the rationale behind the pattern template. It should explain, at a high level, what kind of computation is supported by the template.

B.2 Motivation

This section briefly presents a problem that can be solved using this template. Unlike other patterns documentation, this section does not sketch out the object structure of a solution. However, it may include some details on the decomposition of the problem into parts that can be executed in parallel.

The motivation may include some discussion on the more general pattern structure than is supported by the pattern template. If this is done, then the documentation must state exactly which subset of the pattern structures can be created with the template.

B.3 Applicability

This section describes scenarios in which the pattern template is applicable.

B.4 Design Pattern Template Parameters

This section describes the parameters that can be set to specialize the pattern template, to make it more specific to the problem being solved.

B.5 Framework

This section describes the framework generated for the pattern template from the perspective of the Patterns Layer. It does not include any details on the structure of the framework. This section is broken down into several subsections:

Structure This section shows a simplified object diagram of the framework that shows only those classes that are accessible to the user at the Patterns Layer.

Participants This section lists the classes and/or objects that are part of the framework, and describes their responsibilities.

Collaborations This section describes the interactions between the participants.

Hook Methods The hook methods provided by the framework are described in detail.

Using the Framework This section describes how to instantiate the framework and launch the computation.

Sample Code A simple example is described in detail. The values for the pattern template parameters are listed, and the user code for the application is provided.

B.6 Known Uses

This section lists some real–life uses of the pattern template.

Appendix C

CO_2P_3S Design Pattern Template Documentation

This appendix gives a concrete example of the documentation of a CO_2P_3S design pattern template. The example in the Two–Dimensional Mesh template. This documentation was distributed to the subjects in the usability experiment desribed in Chapter 6.

C.1 Two–Dimensional Mesh Design Pattern Template

C.1.1 Introduction

This report details the Two–Dimensional Mesh design pattern template for CO_2P_3S[1] [67, 68]. This documentation is based on the pattern description format from Gamma *et al.* [37], but we diverge from this format in several ways. The main reason for these changes is that this document is not intended to fully describe a Two–Dimensional Mesh design pattern. Instead, this document shows how to use the Two–Dimensional Mesh design pattern template (also referred to as the Mesh) and how to write an application using the frameworks generated at the Patterns Layer in CO_2P_3S. Given that the main benefit of the Patterns Layer in CO_2P_3S is the encapsulation of the structural code for the generated frameworks, this document does not describe any implementation details beyond those needed to use the frameworks. However, many of the motivational section are needed to know when to apply this pattern template, so they still appear.

It is important to note that a design pattern template is a construct that is derived from a design pattern. The pattern template fixes parts of the implementation of a pattern. Like the patterns they are based on, though,

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

pattern templates represent a family of solutions to a given design problem. The pattern describes this family by presenting implementation options, much like an abstract data type has different implementations. The pattern template provides a family of solutions through a set of template parameters that can be used to select common alternatives and options. These affect the framework code generated for the template. The differences between these two constructs is important – the two terms are not interchangeable, and you should pay close attention to which is being referred to in this document.

Currently, CO_2P_3S pattern templates generate multi–threaded Java framework code targeted at multiprocessor computers with shared memory. While the implementation of the frameworks may take advantage of this architecture, these structural decisions have no effect on the use of the templates or the application code written by the user for the framework.

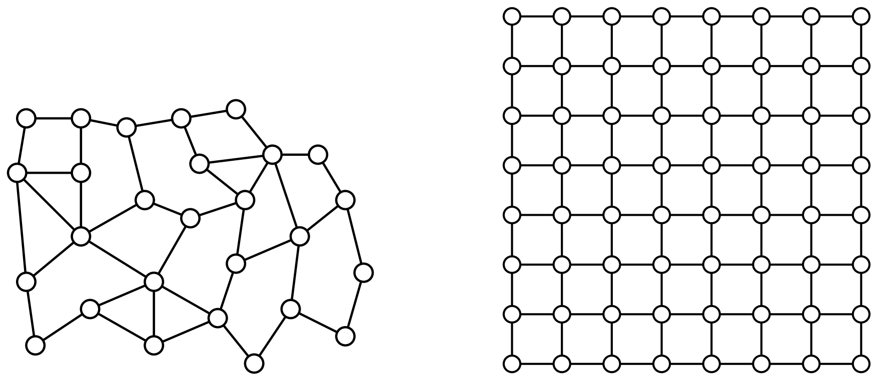
The generated frameworks work as normal object–oriented frameworks. The structure of a complete application, including all classes and the flow of control through them, is defined by a set of abstract classes. These classes are implemented in terms of *primitive methods* or *hook methods* that are invoked by this structural code. A programmer subclasses the abstract classes and provides an implementation of these primitive methods, changing the application while reusing the structure. This is the opposite of a library–based approach to building applications, where the programmer provides the structure of the application and the library provides the primitive methods. A framework provides design reuse by clearly separating the application–independent structure from the application–specific code. The use of frameworks can reduce the effort needed to develop applications by allowing the programmer to concentrate on the application–specific code since the structure is provided.

C.1.2 Intent

The Two–Dimensional Mesh pattern template supports mesh computations on regular, rectangular two–dimensional data. These computations use neighbouring data (also called a *stencil*) to calculate new data values. The Mesh template supports a four–point stencil (using the neighbours on the four compass points) or an eight–point stencil (also including neighbours along the diagonal). The program can use either Jacobi or Gauss–Seidel iteration, but must use local termination conditions.

C.1.3 Motivation

Mesh computations are common in parallel programming. Applications such as weather forecasting and particle simulations can be parallelized using this approach. In the most general two–dimensional case, the mesh is defined as a graph, shown in Figure C.1(a). The nodes of the graph are the *elements* of the mesh, which contain the data values that are to be computed, and the edges show which elements interact with one another. Alternatively, a more regular



(a) An example of a general mesh. (b) An example of a regular mesh.

Figure C.1: Examples of both a general and a regular, rectangular mesh.

structure can be used for many problems, which simplifies both the structure of the mesh data and the implementation of the computation. Such a regular mesh is shown in Figure C.1(b).

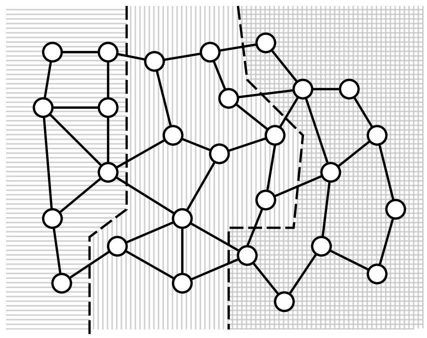
A typical mesh computation repeatedly iterates over the mesh elements and computes new values for each one. This new value is a function of the value in the current element and the values in the elements in some neighbourhood around it. This continues until the final value for the elements is reached.

To parallelize this kind of application, the mesh data is usually spatially decomposed into a set of distinct partitions. Example partitions for the two meshes from Figure C.1 are shown in Figure C.2. Each partition is assigned to a processing unit that computes new values for the mesh elements in its partition. The communication structure of the resulting program is defined by the need for the partitions to exchange their boundaries to compute new values for the elements on the edge of the partition. The structure of a complete parallel mesh program is shown in Figure C.3. This brief description does not even begin to address the issues involved in writing a program to implement such a computation. Such a description is beyond the scope of this document.

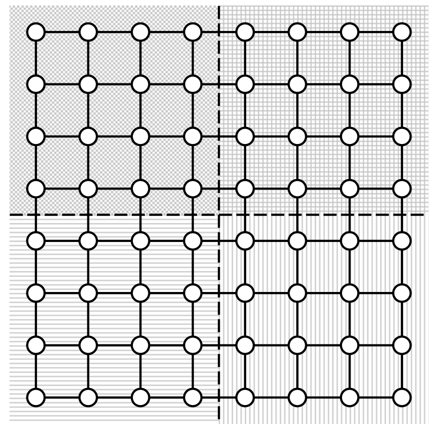
The Two–Dimensional Mesh design pattern template supported by CO_2P_3S simplifies the creation of parallel mesh programs for regular, rectangular mesh data.

C.1.4 Applicability

The Two–Dimensional Mesh pattern template is applicable to problems that consist of elements evenly spaced over a rectangular two–dimensional surface or plane. Each element should use the values from neighbouring elements when calculating a new value, and should be able to determine when it has reached its final value based only on its own state. Aside from requiring the



(a) An example of a decomposed general mesh.



(b) An example of a decomposed regular mesh.

Figure C.2: Example decomposition of a general and a regular, rectangular mesh.

```
Part 1: Create the mesh data.
        Decompose the mesh data into a set of partitions.
        Distribute the partitions over a set of
            processing units.
Part 2: Each processing unit executes the following loop:
            Preprocess data if necessary.
            While ( computation has not finished )
                Preprocess data before use.
                Exchange the boundary between
                    adjacent partitions.
                Compute the new value for the local
                    mesh elements.
            End while
            Postprocess data if necessary.
Part 3: Merge the results.
```

Figure C.3: Pseudocode for the structure of a parallel mesh computation.

values from neighbouring elements, there must not be any other dependencies between the elements (for example, there can be no dependencies that require that new values for the elements be computed in a particular order).

C.1.5 The Mesh Pattern Template Parameters

After selecting the Mesh pattern template from the CO_2P_3S graphical user interface, you must specify the template parameters. These parameters allow some aspects of the framework to be specialized for your application. The parameters for the Mesh template, and their default values when applicable, are:

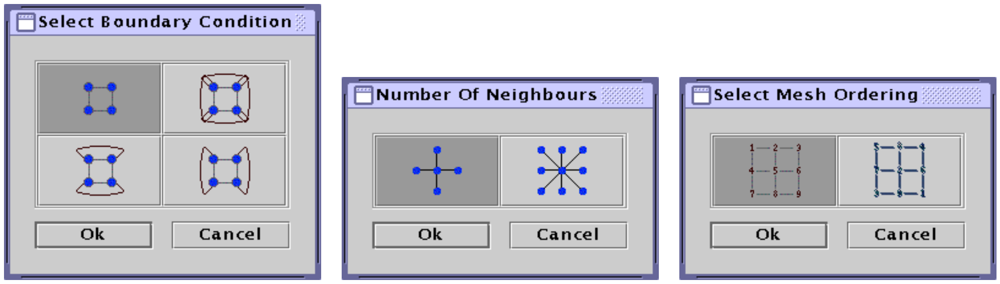
Mesh Class Name This is the name of the class that represents the entire mesh. You create and use instances of this class to create and execute a computation.

Mesh Element Class Name This is the name of the class whose instances are the mesh elements that populate the two–dimensional data for the computation. This may also be referred to as the mesh state class.

Mesh Element Superclass Name This is the name of the class that is the application–specific superclass of the Mesh Element class. This allows the class to be fit into an existing inheritance hierarchy. The default, for Java code, is `Object`.

Boundary Conditions The boundary conditions, or *topology*, indicates how mesh elements on the edges of the mesh data are to be handled. Remember, though, that the data is still two–dimensional. The options supported by the Mesh template are shown in Figure C.4(a). These are (left to right, top to bottom):

- **Non–toroidal** None of the edges wrap around. This is the default value.
- **Fully–toroidal** All edges wrap around to the opposite edge, so each element has access to all neighbours. The two–dimensional data can be treated as a torus or donut.
- **Horizontal–toroidal** The horizontal edges of the data wrap around. Elements on the left and right edges (except the corners) will have access to all neighbours, but the elements on the top and bottom edges (including the corners) will be missing the top and bottom neighbour (respectively). The data can be treated as a cylinder.
- **Vertical–toroidal** Similar to horizontal–toroidal except that the vertical edges wrap around.



(a) The available options for the boundary conditions.

(b) The available options for the mesh stencil.

(c) The available options for mesh ordering.

Figure C.4: Dialogs from CO_2P_3S for specifying some of the Mesh pattern template options.

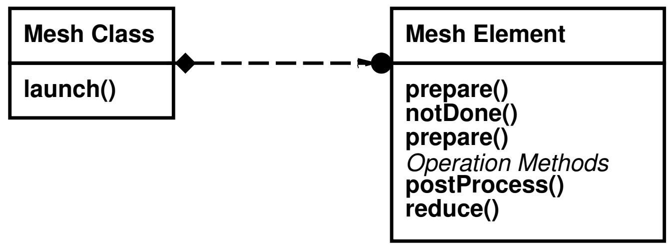
Number of Neighbour Elements This parameter defines the stencil that is used to compute new values for a mesh element (barring missing neighbours because of the selected boundary conditions). The two options supported by CO_2P_3S, a four–point and an eight–point stencil, are shown in Figure C.4(b). The default is the four–point stencil.

Mesh Ordering The mesh ordering indicates the amount of synchronization in the framework code generated for the template. The two available options, *ordered* and *chaotic*, are shown in Figure C.4(c). This synchronization is one of the necessary parts of choosing between Jacobi iteration (an ordered mesh) and Gauss–Seidel iteration (a chaotic mesh). Jacobi iteration also has restrictions on which data is used to compute new values, which must be addressed in the application code. However, this allows intermediate forms of iteration to be implemented. The default value for this parameter is an ordered mesh.

C.1.6 The Mesh Framework

After the parameters to the Mesh pattern template have been supplied, the template is used to generate framework code implementing the selected structure. At the Patterns Layer of CO_2P_3S, the structural parts of the code are hidden from the user. Only those classes that are germane to writing an application can be accessed. This section explains the responsibilities of these classes and shows how they can be used to create a working mesh application.

Structure



Participants

Mesh Element Instances of this class populate the two–dimensional mesh data structure. This class implements the following operations for an individual mesh element:

- creating a single instance by applying an initializer object,
- evaluating the termination condition,
- implementing the mesh operation, and
- gathering the final result by applying a reducer object.

Mesh Class You use this class to create and execute your mesh computation. The class is responsible for:

- creating the two-dimensional mesh data structure and populating it with mesh element objects,
- creating and starting the threads that execute the mesh computation,
- partitioning the data structure and distributing the partitions to the threads,
- evaluating the termination conditions for each iteration of the computation, stopping the computation when all mesh elements have finished, and
- gathering the final results and returning them to the user.

Most of these responsibilities are implemented by iterating over the mesh elements and invoking the appropriate operation for each one. The complete mesh computation, handled by this class, is built up from the individual operations in the Mesh Element class.

Collaborations

The mesh application is created and executed using the Mesh Class. The application is defined by operations implemented for the individual mesh elements in the Mesh Element class. The Mesh Class uses these individual operations to execute a mesh computation over the entire mesh data.

Hook Methods

Each of the three parts from Figure C.3 have different hook methods. Each of these hook methods described here are defined for the Mesh Element class. In CO_2P_3S, these methods are accessed using the template viewers that are available via the context–sensitive pop–up menus on the mesh nodes (after the framework code has been generated, of course).

Hook Methods for Part 1 From Figure C.3 In the first part, the hook method is the constructor for a single mesh element. This constructor is used by the structural code to create the complete mesh data.

The signature for the constructor for a mesh element of type `MeshElem`[2] is:

```
public MeshElem(int i, int j, int dataWidth, int dataHeight,
                Object initializer) ;
```

The arguments to the constructor are:

`i` **and** `j` These two arguments are the indices of the mesh element in the two–dimensional data structure. These values are provided so that location–specific initialization can be done. For instance, the values along the border of the data structure may be set to specific, constant values.

`dataWidth` **and** `dataHeight` These two arguments are used in conjunction with the previous arguments. These are the size of the full mesh data. More precisely, the index `i` ranges over $[0, \texttt{dataWidth} - 1]$ and index `j` ranges over $[0, \texttt{dataHeight} - 1]$.

`initializer` This parameter is a general object supplied to the constructor of the mesh class (discussed further in Section C.1.6, "Using the Mesh Framework", page 202). This object is applied to each mesh object as it is created. For example, the user can pass in a random number generator to initialize the mesh data to random values, or a stream object to read data from a file. This object must be downcast to a more appropriate type before it can be used in the constructor.

The mesh elements are stored in a two–dimensional array and are constructed in the following order for indices [i][j] (where `width` and `height` refer to `dataWidth` and `dataHeight`):

$$
\begin{array}{llll}
[0][0] & [0][1] & \ldots & [0][\texttt{height}-1] \\
[1][0] & [1][1] & \ldots & [1][\texttt{height}-1] \\
 & \ldots & & \\
[\text{width}-1][0] & [\texttt{width}-1][1] & \ldots & [\texttt{width}-1][\texttt{height}-1]
\end{array}
$$

[2]In your program, `MeshElem` would be replaced with the mesh element class supplied in the Mesh template parameters.

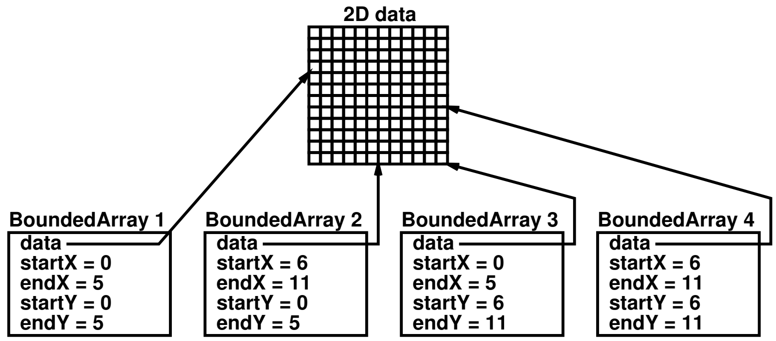


Figure C.5: The use of bounded arrays to share a single copy of an array.

This order is specified so that the initial values can be input from a file in the correct order.

It is also important to make a few notes on how the mesh data is stored, particularly with regard to how the partitions are stored and distributed. As already noted, the mesh data is stored in a two–dimensional array rather than as a graph. However, the frameworks generated by CO_2P_3S also take advantage of the shared memory model. Rather than making a separate copy of each partition for each thread in the computation, the frameworks use the idea of a bounded array, shown in Figure C.5. The instances of the bounded array share a common (usually large) array and have instance variables that define the subarray that they can access. The bounded array then provides a set of accessor methods that allow access to the subarray using local indices (ranging from 0 to the size of the subarray) and translate these requests to global indices. This obviates the need for making copies of the array for each partition, so larger problems can be solved. Further, there is no need to copy data for the boundary exchange; a thread can be allowed to read outside of its partition and access this data directly.

Bounded arrays are not without their drawbacks, though. You must be more aware of the concurrency in your application. This is especially true when implementing a problem using Jacobi iteration. This requires that both the current value (computed during the current iteration) and the previous value (computed during the previous iteration and read by other mesh elements during the current iteration) be kept for each mesh element. Further, the previous value must be updated at the beginning of each iteration. However, these two values must be retained if the problem iterates until the data converges to a final result. The generated framework makes it easy to retain and update these values correctly while still giving enough flexibility to create solutions that use other iteration styles that do not require such careful handling of the data.

Hook Methods for Part 2 From Figure C.3 The second part of a mesh computation is the loop executed by each thread. This loop is responsible

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

Figure C.6: The main loop for each thread in the Mesh framework.

for properly computing new values for the mesh elements. The code for each thread is shown in Figure C.6. In general, each method in the figure is implemented by iterating over the mesh elements in the local partition and invoking the identically named method on each element. Note that the order in which the methods are invoked on the elements of a partition is not defined. The two exceptions are `synchronize()` and `operate()`. The `synchronize()` method implements the synchronization dictated by the Mesh template parameter. For an ordered mesh, this method calls a barrier. For a chaotic mesh, this method does nothing. The `operate()` method invokes one of several methods on the mesh element, as explained below.

The hook methods for this part of the computation, including signatures, are listed below. The default implementation of each of these methods, unless otherwise noted, is empty. Again, all of these methods are defined for the mesh element class.

`public void initialize()` This method is used to implement any preprocessing of the mesh elements that can be done in parallel.

`public boolean notDone()` This method evaluates the termination condition for a single element. A return value of `false` indicates that the computation for the given element has finished, and `true` indicates that the element requires further iterations. Each thread iterates over its local partition to determine if the elements have finished and then exchanges this information with the other threads to determine if the entire computation has finished. The global termination decision is the return value of the `notDone()` method in Figure C.6. All threads continue computing new values for all of the elements in their partition until all elements return `false`. This method returns `false` by default so that the framework will not execute the loop body and the computation will end[3].

[3] By doing this, a framework with the default hook method bodies can be compiled and run immediately after it has been generated, without having to write any code.

Note that the invocations to this method for a given partition are short–circuited; if an element in the partition returns `true`, the iteration stops and the assigned thread reports that it has not finished computing. Thus, it is important that this method not be used as another form of preprocessing as there is no guarantee that it will be invoked on each mesh element in each iteration (except the last iteration, of course).

The synchronization level of the mesh also affects the evaluation of the termination condition. An ordered mesh adds barrier synchronization when the threads exchange their termination status. This ensures that all of the threads finish evaluating the termination conditions for a given iteration before computing new values. In a chaotic mesh the barriers are not used, so a thread may be using the termination status of an older or more recent iteration. This makes it difficult to be sure that the mesh data has converged to its final answer, so extra measures must usually be taken to ensure that the termination conditions have been met.

Finally, be aware that the termination condition is checked before the first iteration of the computation. The initial conditions for the mesh element must ensure that the termination condition will not be true the first time the condition is checked.

`public void prepare()` This method is used for any preprocessing required in each iteration of the computation. If the mesh is ordered, each thread will finish this preprocessing before any thread begins to compute new values (because of the barrier used in the implementation of `synchronize()` in Figure C.6). This method can be used to update the read value before the next iteration begins to compute new data values.

The operation methods These methods define the mesh operation for a single mesh element. There are up to nine different methods that can be invoked on a given element, depending on its location in the mesh data and the selected boundary conditions. Further, the signatures of the methods vary depending on the choice of a four–point or an eight–point stencil. The complete list of operation methods for a four–point stencil is shown is Figure C.7, and the list of operation methods for an eight–point stencil is in Figure C.8.

The structural framework code is responsible for determining the correct operation method to invoke and for supplying the correct neighbouring mesh elements for the arguments. The operation methods for the different elements for each boundary condition are shown in Figure C.9. This figure shows the calls for the complete mesh data; a given partition will call the relevant methods for the subset of elements that have been assigned to it. To make using the framework easier, CO_2P_3S generates stubs for each relevant operation method. This means that you do not have to determine either the signatures or the correct set of operation methods.

```
public void topLeftCorner(MeshElem right, MeshElem down)
public void topEdge(MeshElem left, MeshElem right,
            MeshElem down)
public void topRightCorner(MeshElem left, MeshElem down)
public void leftEdge(MeshElem right, MeshElem up,
            MeshElem down)
public void interiorNode(MeshElem left, MeshElem right,
            MeshElem up, MeshElem down)
public void rightEdge(MeshElem left, MeshElem up,
            MeshElem down)
public void bottomLeftCorner(MeshElem right, MeshElem up)
public void bottomEdge(MeshElem left, MeshElem right,
            MeshElem up)
public void bottomRightCorner(MeshElem left, MeshElem up)
```

Figure C.7: Signatures for the operation methods for a four–point mesh.

`public void postprocess()` This method, the complement to `initialize()`, is used to implement any postprocessing of the mesh elements that can be done in parallel. For instance, this method can update the read value of a mesh element one last time, before the results are gathered.

Hook Methods for Part 3 From Figure C.3 The last part of the mesh computation is to gather the results so that they can be returned to the user. The framework implements this gathering as a sequential operation that is performed after the computation, as there is no way to be certain that it can be safely done in parallel.

The gather operation operates in a similar fashion to the constructor for the mesh element from Section C.1.6 ("Hook Methods for Part 1 From Figure C.3", page 195). The signature for this method, defined on the mesh element class, is:

```
public void reduce(int i, int j, int dataWidth, int dataHeight,
                   Object reducer) ;
```

Note that this is not a proper parallel reduction, but rather reduces the final mesh data to a single object that can be returned as the result of the program.

The arguments to this method are:

`i` **and** `j` These two arguments are the indices of the mesh element in the two–dimensional data structure. These values are provided so that location–specific gathering can be implemented. For instance, if the values along the edges of the mesh data are constant and not part of the overall solution, they may not be gathered.

```
public void topLeftCorner(MeshElem east, MeshElem southeast,
            MeshElem south)
public void topEdge(MeshElem east, MeshElem southeast,
            MeshElem south, MeshElem southwest, MeshElem west)
public void topRightCorner(MeshElem south, MeshElem southwest,
            MeshElem west)
public void leftEdge(MeshElem north, MeshElem northeast,
            MeshElem east, MeshElem southeast, MeshElem south)
public void interiorNode(MeshElem north, MeshElem northeast,
            MeshElem east, MeshElem southeast, MeshElem south,
            MeshElem southwest, MeshElem west,
            MeshElem northwest)
public void rightEdge(MeshElem north, MeshElem south,
            MeshElem southwest, MeshElem west,
            MeshElem northwest)
public void bottomLeftCorner(MeshElem north,
            MeshElem northeast, MeshElem east)
public void bottomEdge(MeshElem north, MeshElem northeast,
            MeshElem east, MeshElem west, MeshElem northwest)
public void bottomRightCorner(MeshElem north, MeshElem west,
            MeshElem northwest)
```

Figure C.8: Signatures for the operation methods for a eight–point mesh.

topLeftCorner()	topEdge()	. . .	topEdge()	topRightCorner()
leftEdge()	interiorNode()		interiorNode()	rightEdge()
. . .	. . .	. . .		. . .
leftEdge()	interiorNode()		interiorNode()	rightEdge()
bottomLeftCorner()	bottomEdge()	. . .	bottomEdge()	bottomRightCorner()

(a) Non–toroidal operations.

interiorNode()	interiorNode()	. . .	interiorNode()	interiorNode()
interiorNode()	interiorNode()		interiorNode()	interiorNode()
. . .	. . .	. . .	. . .	. . .
interiorNode()	interiorNode()	. . .	interiorNode()	interiorNode()
interiorNode()	interiorNode()		interiorNode()	interiorNode()

(b) Fully–toroidal operations.

topEdge()	topEdge()	. . .	topEdge()	topEdge()
interiorNode()	interiorNode()		interiorNode()	interiorNode()
. . .	. . .	. . .	. . .	. . .
interiorNode()	interiorNode()	. . .	interiorNode()	interiorNode()
bottomEdge()	bottomEdge()		bottomEdge()	bottomEdge()

(c) Horizontal–toroidal operations.

leftEdge()	interiorNode()	. . .	interiorNode()	rightEdge()
leftEdge()	interiorNode()	. . .	interiorNode()	rightEdge()
. . .	. . .	. . .	. . .	. . .
leftEdge()	interiorNode()	. . .	interiorNode()	rightEdge()
leftEdge()	interiorNode()	. . .	interiorNode()	rightEdge()

(d) Vertical–toroidal operations.

Figure C.9: The operation method calls for mesh elements in different positions, for each boundary condition from Figure C.4(a).

`dataWidth` and `dataHeight` These two arguments are used in conjunction with the previous arguments. These are the size of the full mesh data. More precisely, the index `i` ranges over $[0, \texttt{dataWidth} - 1]$ and index `j` ranges over $[0, \texttt{dataHeight} - 1]$.

`reducer` This parameter is a general object supplied to the constructor to the mesh class (see Section C.1.6, "Using the Mesh Framework", page 202). It is applied to each mesh element, and is returned to the user as the result of the mesh computation. Thus, this object will be modified via side effects to hold the final value. For example, the reducer can be a container object into which the data is copied. It may also be an accumulator so that the sum of the mesh data can be returned.

The gather is performed in the following order for indices [`i`][`j`] (where `width` and `height` refer to `dataWidth` and `dataHeight`):

$[0][0]$	$[0][1]$	...	$[0][\texttt{height} - 1]$
$[1][0]$	$[1][1]$	...	$[1][\texttt{height} - 1]$
	...		
$[\text{width} - 1][0]$	$[\texttt{width} - 1][1]$	...	$[\texttt{width} - 1][\texttt{height} - 1]$

Like the constructor, this order is provided so that the results can be written to a file if the reducer is a stream object.

Using the Mesh Framework

The last step in writing an application using the Mesh framework is to instantiate the computation and execute it. As you might expect, this is done by creating and using an instance of the mesh class.

The constructor for the mesh class `MeshClass`[4] is:

```
public MeshClass(int dataWidth, int dataHeight,
                 int meshWidth, int meshHeight,
                 Object initializer, Object reducer) ;
```

The arguments to the constructor are:

`dataWidth` and `dataHeight` These two arguments are the size of the mesh data. The indices for the two–dimensional array that will hold the data will range from $[0][0]$ to $[\texttt{dataWidth} - 1][\texttt{dataHeight} - 1]$.

`meshWidth` and `meshHeight` These arguments are the number of partitions in the horizontal and vertical direction respectively. The mesh data is always decomposed into rectangular partitions. A separate thread will be created for each partition. The number of partitions should be at

[4] In your program, `MeshClass` would be replaced with the mesh class name supplied in the Mesh template parameters.

least the number of processors that you want to use. It is also possible to create more threads than processors; you will need to experiment on your machine to determine the best ratio of threads to processors.

`initializer` This is the initializer object that is applied in the constructor for each mesh element. Any type of object can be used.

`reducer` This is the reducer object that is applied during the gathering of results. Any type of object can be used. The final result is placed into this object by side effects in the gather part of the computation.

To execute the mesh computation, the method

```
public void launch() ;
```

should be invoked on the instance of `MeshClass`. This method returns when the mesh computation has completed. The final results will be available in the `reducer` object.

Sample Code

As an example of the use of the Mesh pattern template and generated framework, we will show a specification and implementation of the Game of Life.

The game is a simple cellular automata simulation. A group of cells on a two–dimensional surface live and die based on the following rules:

1. If a cell is dead but has three neighbours, then it is brought to life.

2. If a cell is alive and has two or three neighbours, it survives. If it has less than two neighbours, it dies of loneliness. If the cell has more than three neighbours, it dies of overcrowding.

This particular version of the game uses Jacobi iteration on a non–toroidal surface. The initial state of the cells is determined randomly. The simulation continues for a fixed number of iterations, and the state of the cells is drawn to a graphical window.

For this problem, the following parameter values are used for the Mesh pattern template:

Mesh Class Name Set to `LifeMesh`.

Mesh Element Class Name Set to `LifeElem`.

Mesh Element Superclass Name Left with the default, `Object`.

Boundary Conditions Set to non–toroidal.

Number of Neighbour Elements Set to an eight–point stencil.

Mesh Ordering Set to an ordered mesh.

`LifeElem` Class Implementation The implementation of the `LifeElem` class follows. There are some things to note about the `LifeElem` code. First, note the use of a read and write value for each element. The new state of a cell is computed based on the read value. However, the read value must be updated during each iteration, before the new states are calculated. This is done in the `prepare()` method.

```java
import java.lang.* ;
// Enter additional imports here.
import java.util.Random ;

public class LifeElem
    extends Object
{
    public LifeElem(int i, int j, int surfaceWidth, int surfaceHeight,
                Object initializer)
    {
        Random generator = (Random) initializer ;
        this.setValue((int) (generator.nextDouble() + 0.5)) ;
    } /* LifeElem */

    public void initialize()
    {
        this.updateReadValue() ;
    } /* initialize */

    public boolean notDone()
    {
        return(this.getIteration() <= MaxIterations) ;
    } /* notDone */

    public void prepare()
    {
        this.updateReadValue() ;
        this.incrementIteration() ;
    } /* prepare */

    public void interiorNode(LifeElem north, LifeElem northeast, LifeElem east,
                        LifeElem southeast, LifeElem south, LifeElem southwest,
                        LifeElem west, LifeElem northwest)
    {
        int sum = north.getValue() + northeast.getValue() + east.getValue() +
                southeast.getValue() + south.getValue() + southwest.getValue() +
                west.getValue() + northwest.getValue() ;
        this.evaluateCell(sum) ;
    } /* interiorNode */


    public void topEdge(LifeElem east, LifeElem southeast, LifeElem south,
                    LifeElem southwest, LifeElem west)
    {
        int sum = east.getValue() + southeast.getValue() + south.getValue() +
```

```
                southwest.getValue() + west.getValue() ;
    this.evaluateCell(sum) ;
} /* topEdge */


public void bottomEdge(LifeElem north, LifeElem northeast, LifeElem east,
                       LifeElem west, LifeElem northwest)
{
    int sum = north.getValue() + northeast.getValue() + east.getValue() +
              west.getValue() + northwest.getValue() ;
    this.evaluateCell(sum) ;
} /* bottomEdge */


public void leftEdge(LifeElem north, LifeElem northeast, LifeElem east,
                     LifeElem southeast, LifeElem south)
{
    int sum = north.getValue() + northeast.getValue() + east.getValue() +
              southeast.getValue() + south.getValue() ;
    this.evaluateCell(sum) ;
} /* leftEdge */


public void rightEdge(LifeElem north, LifeElem south, LifeElem southwest,
                      LifeElem west, LifeElem northwest)
{
    int sum = north.getValue() + south.getValue() + southwest.getValue() +
              west.getValue() + northwest.getValue() ;
    this.evaluateCell(sum) ;
} /* rightEdge */


public void bottomLeftCorner(LifeElem north, LifeElem northeast, LifeElem east)
{
    int sum = north.getValue() + northeast.getValue() + east.getValue() ;
    this.evaluateCell(sum) ;
} /* bottomLeftCorner */


public void topLeftCorner(LifeElem east, LifeElem southeast, LifeElem south)
{
    int sum = east.getValue() + southeast.getValue() + south.getValue() ;
    this.evaluateCell(sum) ;
} /* topLeftCorner */


public void bottomRightCorner(LifeElem north, LifeElem west, LifeElem northwest)
{
    int sum = north.getValue() + west.getValue() + northwest.getValue() ;
    this.evaluateCell(sum) ;
} /* bottomRightCorner */


public void topRightCorner(LifeElem south, LifeElem southwest, LifeElem west)
```

```
{
    int sum = south.getValue() + southwest.getValue() + west.getValue() ;
    this.evaluateCell(sum) ;
} /* topRightCorner */

public void postProcess()
{
    this.updateReadValue() ;
} /* postProcess */

public void reduce(int i, int j, int surfaceWidth, int surfaceHeight, Object reducer)
{
    LifeDisplayWindow win = (LifeDisplayWindow) reducer ;
    win.setPixel(i, j, this.getValue()) ;
} /* reduce */

// Enter additional user code here.
// Apply the rules of the Game of Life based
// on the number of alive neighbours.
public void evaluateCell(int numNeighbours)
{
    // If the cell is already alive...
    if (this.getValue() == 1) {
        if (numNeighbours < 2 || numNeighbours > 3) {
            this.setValue(0) ;
        } /* if */

    // Else, if the cell was dead...
    } else {
        if (numNeighbours == 3) {
            this.setValue(1) ;
        } /* if */
    }  /* if */
} /* evaluateCell */


// Value read by neighbours.
public int getValue()
{
    return(this._readValue) ;
} /* getValue */

// Update the read value with the write value.
public void updateReadValue()
{
    this.setReadValue(this.getWriteValue()) ;
} /* updateReadValue */

// Set the current value (the write value).
// Only used by the current element.
protected void setValue(int value)
{
    this._writeValue = value ;
} /* setValue */
```

```
    protected int getWriteValue()
    {
        return(this._writeValue) ;
    } /* getWriteValue */

    protected void setReadValue(int value)
    {
        this._readValue = value ;
    } /* setReadValue */


    protected int getIteration()
    {
        return(this._iterations) ;
    } /* getIterations */

    protected void incrementIteration()
    {
        ++this._iterations ;
    } /* incrementIteration */


    // Instance variables.
    private int _readValue ;
    private int _writeValue ;
    private int _iterations = 0 ;

    // Constants.
    private static final int MaxIterations = 100 ;
} /* LifeElem */
```

Mainline Class Implementation The mainline for the complete application follows. The only thing to note about this class in that all of the sizes are fixed. In a real program, some of the values would be provided on the command line. Otherwise, changing the size of the mesh data or the number of threads to use requires the class to be modified and recompiled.

```
import java.lang.* ;
import java.util.Random ;

public class LifeMain
{
    protected final static int SurfaceSize = 100 ;
    protected final static int NumberPartitions = 2 ;

    public static void main(String[] args)
    {
        // Create the initializer and reducer.
        Random initializer = new Random(1) ;
```

```
        LifeDisplayWindow reducer = new LifeDisplayWindow(SurfaceSize,
                                                           SurfaceSize) ;

        // Create the mesh and execute the computation.
        LifeMesh mesh = new LifeMesh(SurfaceSize, SurfaceSize,
                                     NumberPartitions, NumberPartitions,
                                     initializer, reducer) ;
        mesh.launch() ;

        // The data has been gathered into the display window.
        // Show it now.
        reducer.show() ;

        System.out.println("Program finished.") ;
    } /* main */
} /* LifeMain */
```

C.1.7 Known Uses

The Two–Dimensional Mesh pattern template can be used for a number of applications. There are many variations of programs with structures like the LaPlace solver, including some image processing applications such as image skeletonization (thinning all line segments in an image to be a single pixel wide while preserving the connectivity of the original line segments, typically done as the first step in handwriting analysis), that can be solved using this pattern template.

Appendix D

Material for the Usability Experiment

This appendix contains the descriptions of the three assignments that made up the usability experiment. In the course, they were the third, fourth, and fifth assignments. Also included is any additional documentation given to the students for the assignment. The pattern template description for the Mesh, in Appendix C.1, was also distributed as part of the assignments.

Note that Assignment 5 is a version of the reaction–diffusion program with the terminology changed (the assignment refers to a mould and bacteria spreading over the surface of a donut rather than morphogens on an unspecified surface).

The "Mesh Computations" document referenced in Section D.1 can be found in Appendix E. The "Parallel Mesh Computations" document referenced in Section D.2 is in Appendix F.

D.1 Assignment 3: Sequential Thermal Computation

D.1.1 Assignment Description

Read the "Mesh Computations" document to learn about computations on meshes. Then solve the following problem in Java

Consider a rectangular piece of alloy consisting of three different metals, each with different thermal characteristics. For each region of the alloy, there is a fixed amount (expressed in terms of a percentage) of each of the three base metals. The top left corner (at the mesh element at index [0,0]) is heated at S degrees Celsius and the bottom right corner (index [width - 1,height - 1]) is heated at T degrees Celsius. The temperature at these points is constant and does not change.

Your assignment is to calculate the final temperature for each region on the piece of alloy. The new temperature for a given region of the alloy is calculated using the formula:

$$temp = \sum_{m=1}^{3} C_m * \left(\sum_{n \in N} temp_n * p_n^m \right) / |N|$$

where m represents each of the three base metals, C_m is the thermal constant for metal m, N is the set representing the neighbouring regions, $temp_n$ is the temperature of the neighbouring region, p_n^m is the percentage of metal m in neighbour n, and $|N|$ is the number of neighbouring regions. Think about how the neighboring regions should be defined after reading the "Mesh Computations" document. This computation must be repeated until the temperatures converge to a final value or a reasonable maximum number of iterations is reached.

The values for S, T, C_1, C_2, C_3, the height and width of the mesh, and the threshold should be parameters to the program. The percentage of each metal in the alloy at each region can be the output of a random number generator (simply take three random numbers between 0 and 1 and calculate the percentages of each relative to the sum of the three numbers). However, you should fix the initial seed for the generator to 0 so that your results are deterministic and reproducible.

subsectionClarifications

First, as stated above, the parameters to the program should be given on the command line. This will probably be easiest for all involved. Make sure you read the list of parameters carefully so your program accommodates all of them. There are 8 in total.

Second, I have found that, although the assignment doesn't mention this, setting the initial temperature of the mesh data to a random value between S and T provides good results in a reasonable amount of time.

Third, the output of your program should be the final temperatures on the surface. Just print this out to standard out using println(). Each row should be a single line in the output, such as

$$\begin{array}{llll}
[0][0] & [0][1] & \ldots & [0][\texttt{height}-1] \\
[1][0] & [1][1] & \ldots & [1][\texttt{height}-1] \\
 & \ldots & & \\
[\text{width}-1][0] & [\texttt{width}-1][1] & \ldots & [\texttt{width}-1][\texttt{height}-1]
\end{array}$$

Don't worry about the size or readability of the output - it's not intended to be read, but rather it's going to be used to plot a graph, which is why it is important to make sure each row is on a single output line. Also, make sure you don't print any other output in your final version. When you run your program, capture this data by redirecting your program's output to a file:

```
java ThermalMesh 0 1 1.0 1.0 1.0 100 100 0.05 > output
```

To produce a graph, take this small script (between the dashes but not including them) and save it as "gunplot.commands":

```
set terminal png color
set output "plot.png"
set cntrparam linear
set surface
set hidden3d
set dgrid3d width,height
set ticslevel 0
splot "output" matrix notitle with lines
```

Change `width` and `height` in the 6th line to the width and height of your mesh (100,100 for the example above). You can plot the graph with the following command:

```
gnuplot gunplot.commands
```

The plot will be in the file "plot.gif", which you can view using xv or your favourite image viewer.

Finally, the above set of parameters produces a plot that is similar to the one in the "Mesh Computations" document (possibly oriented differently depending on the order in which you print your data). My program takes about 270 iterations to converge. Don't be alarmed if your program takes a different number of iterations. This should be a good check to see if you're on the right track.

If you want to experiment with different parameters, I have a couple of suggestions. The problem has the best chances of converging if all three thermal constants are close to 1.0 and $C_1 * C_2 * C_3$ is around 1.0. You should see fluctuations in the temperature over the surface of the metal as some regions will have larger amounts of particular metals and thus retain more heat than others.

Some other observations: Any thermal constant above 1.0 would be best described as exothermic - it introduces heat into the alloy, which makes convergence less likely (the temperatures may never settle as more heat is being introduced during each iteration of the program). Low values (even as low as 0.9) tend to be rather endothermic - they absorb the heat. With these, the middle portion of the metal has a very low temperature since the heat from the corners dissipates before it can reach these parts of the alloy.

D.1.2 Comments on Design

A number of people have considered taking the example mesh computation from the "Mesh Computations" document and replacing the loop bodies with the computation in the assignment description. Remember that this course is about object-oriented design and programming. You should be applying these practices to this program as well. Some things to consider are:

- How easy would it be to change the problem being solved without changing the structure of the mesh?

- How easy would it be to introduce some of the alternative mesh structures without having to change the mesh element structure?

The example in the "Mesh Computations" document does not address either of these problems. The structure of the mesh computation and the computation itself are tied together. Changing either will require considerable effort, and neither can really be changed individually. Although you may not achieve a complete separation between the two aspects of this program, a partial separation should definitely be possible.

D.2 Assignment 4: Parallel Thermal Computation

D.2.1 Assignment Description

Read the "Parallel Mesh Computations" document to learn about parallel mesh computations. Then solve the following problem in Java.

Your assignment is to write a parallel version of the first assignment. If you wish, you may use this example implementation of the first assignment. If possible, though, use your own assignment. However, you should add two more command line parameters at the end of the parameter list in this order: the number of horizontal partitions and the number of vertical partitions. For example, if you partition a grid of width 12 nodes and height 8 nodes into blocks of 6 by 2 nodes, the number of horizontal partitions would be 4 and the number of vertical partitions would be 2. You could then use 8 threads to solve the problem, one thread for each 6 by 2 block of nodes.

You must also make one other change to Assignment 3. If you initialize the concentrations of the three metals to different random values in different regions, the computation does not converge very well for many parameter values. Therefore, pick a single set of random concentrations for the three metals and use these same concentrations in all regions.

You have been given temporary accounts to work in for this assignment. You will need to use these accounts to access the software needed to complete the assignment.[1]
The password for these accounts is the same as your normal CS account. Note that these accounts are set up for us to make measurements of your development process. Some commands (specifically, the commands for compiling and running Java programs) are not in the usual place. Further, the shell for these accounts is `zsh`. Please do not modify this environment or change shells. The accounts, as set up, should be sufficient for you to do this assignment.

The class has also been broken into two different groups. The 42501 group will implement this assignment using CO_2P_3S , a parallel programming system

[1] The details on the accounts has been removed to protect the privacy of the students involved.

under development here at the university. You will use the Two–Dimensional Mesh design pattern template available in the tool. Read the documentation for the template to help you.

The 42502 group will implement this assignment using straight Java. To help you, an implementation of barrier synchronization is provided. You can read the documentation for details on how to use the library. You do not have to compile this code to use this class, just include an import statement in your code:

```
import EDU.oswego.cs.dl.util.concurrent.CyclicBarrier;
```

The implementation code is provided in case you want to see how Barriers are implemented. You don't have to understand the implementation to use a Barrier.

The groups will switch for the next assignment. That is, the ones using straight Java for this assignment will use COPS for the next assignment and visa versa.

Note there will be two 45 minute demos of the COPS system for those using it for the first assignment. The demos will be in CSC-251 (Software Systems Lab) at 3:00 and 5:00pm on Friday October 13. The demos will be repeated for the second group before they use COPS for the next assignment.

Both groups will use a *native–threaded* Java virtual machine. On such a JVM, the thread library used in the implementation of the `Thread` class will schedule threads onto separate physical processors on a multiprocessor machine. (In contrast, a *green–threaded* JVM will never use more than one processor regardless of the number of threads created. The JVM in the lab does not support green threads, only native threads.)

The final output of your program, as before, should be the final values of the mesh elements. However, unlike the previous assignment where these values were printed to the standard output, these values should be written to a file called "output".

When developing and debugging your program, use a single processor machine (such as the workstations in the lab CSC 1-21). For the two nights before the assignment is due, we have arranged time on ohaton, a 4 processor Sparc server.

As part of your assignment, you will be expected to provide performance measurements of your parallel program. These measurements should include both of the following:

1. The wall clock run time of the computation time of your program. This time should include initialization time for any mesh data, but should not include the output time.

2. The *speedup* of your program. The speedup of a parallel program is computed as
$$S = \frac{T_s(n)}{T_p(n)}$$

where $T_s(n)$ is the wall clock time of your sequential program (again, including initialization time but not output time) on a problem of size n and $T_p(n)$ is the wall clock time of the parallel program using p threads on a problem of size n. Note that both the sequential and parallel programs should be executed on the same machine to ensure consistent results. Try to include performance numbers for 2 and 4 processor runs.

Some tips for obtaining better speedups:

- Run with a larger problem. Simply put, larger problems help amortize the overheads of a parallel program. You may need to increase the heap size of the virtual machine. You can to this by using the command line arguments `-Xms` and `-Xmx` to set the initial and maximum heap sizes. For best results, set both of these to the same value. For example, to set the initial size to 32 MB and the maximum heap size to 64 MB, add the flags `-Xms32m -Xmx64m` to the command line. Make sure to run both the sequential and parallel assignment with the same problem size. However, please bear in mind the limited resources for obtaining performance data. Your program can only run for approximately 30 minutes if everyone is to get a chance to obtain results.

- Remember that synchronization is an overhead. Minimize your use of barriers and synchronized methods where this does not compromise correctness.

- Try to ensure that the amount of computation for each thread in your program is equal. Avoid situations where processors are idle waiting for other processors. For instance, when using barrier synchronization, try to ensure that each processor gets to the barrier at roughly the same time.

- Run your program through a profiler to get an idea of which parts of your program are the most time–consuming. Once these have been identified, you can try to optimize these parts.

D.2.2 Added Comments

- A small mistake in the COPS documentation has been found. On the last line of page 16, the last sentence should read "The final results will be available in the *reducer* object.", as opposed to the initializer object.

- The web page suggests the use of the method `System.getTimeMillis()` to get timing information. The correct method is `System.currentTime-Millis()`. Sorry for the mixup.

- Also, remember that you should measure the times for initialization and computation only, not the time for your final output. This final output should be written to a file using a file stream, rather than just redirecting the output as in the previous assignment.

- A bug has been uncovered in COPS when changing some of the Mesh template parameter values and regenerating the code. Specifically, if you change your mesh from a 4-point stencil to an 8-point stencil and regenerate the First Layer Code (again, generating the Next Layer Code does nothing), some of the classes aren't properly regenerated. This leads to compile errors as the old versions of the classes are using the incorrect signatures for the operation methods in the mesh element class. The workaround for now is to create a new program with the new values for the parameters and re-enter your program code. You can cut and paste code into the editing components in the COPS GUI using Control-C to copy and Control-V to paste. It is not clear if this problem will occur for other parameter changes. (You should not use an 8 point mesh for this assignment. DS)

- I have a few example sets of parameters for Assignment 4. Since the amount of each metal in the alloy is uniform over the whole surface, the results will be similar to the plot on the assignment web page. However, the thermal constants can be used to shift the flat part of the graph up or down. The following three sets of parameters shifted the central part of the graph from a value below the two corners to a value much higher.

  ```
  java Main 10 11 1.0 1.05 0.98 100 100 0.025
  java Main 10 11 1.0 1.05 0.981 100 100 0.025
  java Main 10 11 1.0 1.05 0.982 100 100 0.025
  ```

  You may need to play with these parameters to get these results. You can also change the values by changing the thresholds, particularly for the first two problems. Both of them very slowly absorb heat, so a lower threshold will force more iterations and will slowly reduce the temperature of the metal. In your assignment write up, give the complete set of parameters that you used for your timings.

- You can execute COPS programs from home without starting the GUI. Be warned, though, that you will not be able to edit them.

- If you examine the directory that you placed your COPS program in, you will find that the name of your program corresponds to a directory. Within that directory, there will be a subdirectory called "classes". From this directory, you can run your program from the command line as you would normally.

D.3 Assignment 5: Mould and Bacteria

D.3.1 Assignment Description

This assignment to is write a parallel mesh computation that simulates a mould and bacteria on the surface of a Tim Horton's glazed chocolate donut that has been thrown away. Both the mould and bacteria will grow and spread over the surface of the donut. Some portion of each will also leave the simulated donut. In addition, the mould impedes the growth of the bacteria where the two come into contact.

The simulation involves computing the concentration of both the mould and bacteria over the surface over time. The concentration for a particular participant at a given point on the surface is denoted $C(x, y)$, as the surface is considered to be a two–dimensional surface folded into a torus. When it is important to note which participant is being referred to, the term will be subscripted with a m for the mould or a b for the bacteria.

Numerically, this simulation is the solution of the non–linear partial differential equation

$$\dot{C} = a^2 \left(\frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2} \right) - dC + R \tag{D.1}$$

where $\dot{C}$ is the time derivative of the concentration, a is the rate of at which the participant spreads, d is the rate at which the participants leave the simulation, and R is a function that describes the reaction between the mould and the bacteria.

This problem can be solved by approximating the solution of Equation D.1 using finite differences solved with Euler's method. The problem can now be restated as the repeated application of the formula

$$C_{t+\Delta t} = \Delta t(M \otimes C_t + R_t) \tag{D.2}$$

over time with timestep Δt, where "$\otimes$" is the discrete convolution operator (described in the appendix of this assignment) with the convolution mask M. This formula is applied twice, once for the mould and once for the bacteria. The mask M is the three by three mask for computing the centre element

$$M = \frac{1}{2} \left[\begin{array}{ccc} -a_{12} & 2a_{22} & a_{12} \\ 2a_{11} & -4(a_{11} + a_{22}) - 2d & 2a_{11} \\ a_{12} & 2a_{22} & -a_{12} \end{array} \right] \tag{D.3}$$

where

$$a_{11} = a_1^2 \cos^2 \theta + a_2^2 \sin^2 \theta \tag{D.4}$$

$$a_{12} = (a_2^2 - a_1^2) \cos \theta \sin \theta \tag{D.5}$$

$$a_{22} = a_2^2 \cos^2 \theta + a_1^2 \sin^2 \theta \tag{D.6}$$

where a_1 is the rate of movement along the axis $[\cos\theta, \sin\theta]$, a_2 is the rate of movement along the axis $[-\sin\theta, \cos\theta]$, and θ defines the rotation of the axes.

For a given participant, the convolution mask can be specified as a triple $P = (a_1, a_2, \theta)$. In addition, the function R must also be supplied, which may rely on the concentration values of all participants in the simulation. Each of these should be parameters to the classes that implement your simulation.

Good results for a simulation can be achieved by starting the concentration of both the mould and bacteria to random values in the range $[-1, 1]$ and using the following values for each of the parameters:

$$\Delta t = 0.02 \tag{D.7}$$
$$d = 1.0 \tag{D.8}$$
$$P_m = (2.0, 1.5, \theta) \tag{D.9}$$
$$P_b = (2.0, 1.0, \theta) \tag{D.10}$$

where the value of θ should be the same for both the mould and the bacteria, but otherwise can be any value. The function R is defined as:

$$R_m = R_b = \text{if } \mathrm{C_m} > \mathrm{C_b} \text{ then } 1.0 \text{ else } 0.0 \tag{D.11}$$

Your Solution

Your program should have four command line parameters in this order: the width of the mesh, the height of the mesh, the angle theta, and the output file name. The output file should contain the final concentrations of the bacteria. A program to display the results will be provided.

You must construct a sequential solution and a parallel version. Put each in a separate directory.

The 42501 group will implement this assignment using straight Java. To help you, an implementation of barrier synchronization is provided. You can read the documentation for details on how to use the library. You do not have to compile this code to use this class, just include an import statement in your code:

```
import EDU.oswego.cs.dl.util.concurrent.CyclicBarrier;
```

The implementation code is provided in case you want to see how Barriers are implemented. You don't have to understand the implementation to use a Barrier.

The 42502 group will use COPS to implement this assignment. Read the COPS documentation for the mesh template to help you. Note there will be two 45 minute demos of the COPS system for those using it for this assignment. The demos will be in CSC-251 (Software Systems Lab) at 3:00 and 5:00pm on Monday October 23.

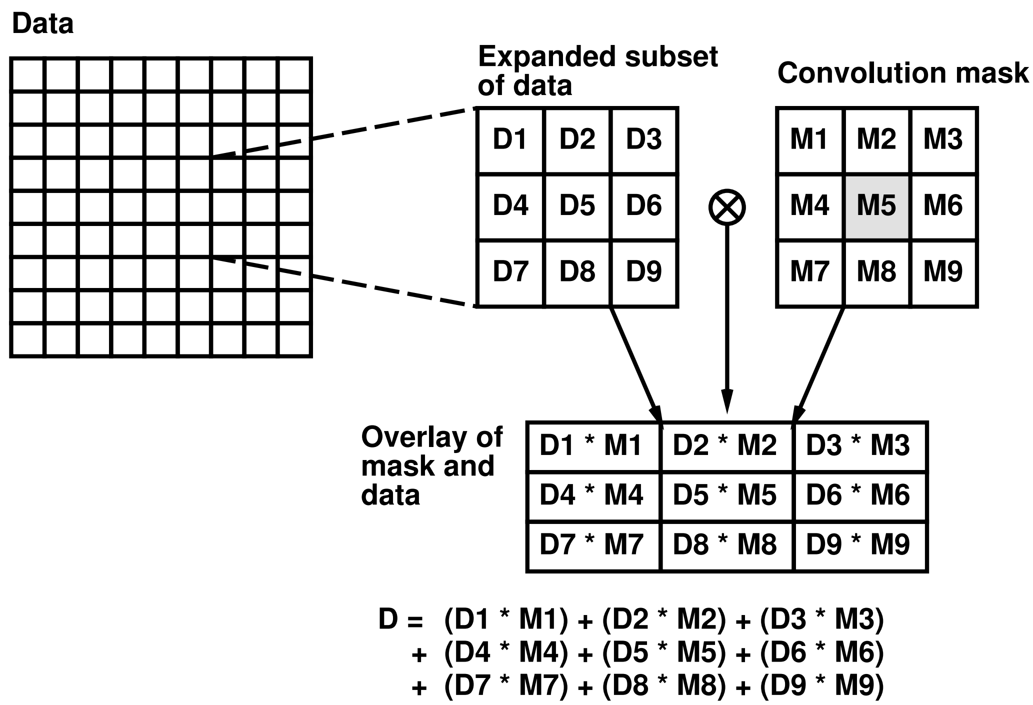


Figure D.1: An example of the convolution of a three by three mask over two–dimensional data. The element that is being computed is the grey element in the centre.

Appendix: Discrete Convolution

Discrete convolution is the process of applying a *convolution mask* to a data set. This mask represents a formula that is a weighted combination of the current element and the elements in a neighbourhood around it. This mask is shifted over the elements of the data set in some order. An example based on a three by three mask is shown in Figure D.1. Typically, the value of $D5$ is replaced with the computed value D. It may help to imagine overlaying the mask on top of the data.

The convolution mask is usually specified using a matrix, where one element in the matrix is flagged as the element whose value is being computed. As a concrete example, the LaPlace solver for the interior region of a data set can be specified as the convolution mask

$$M = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{D.12}$$

D.3.2 Notes and Clarifications

- There is a typo is Equation D.2. It should read

$$C_{t+\Delta t} = C_t + \Delta t(M \otimes C_t + R_t) \tag{D.13}$$

- The function R is also incorrect. It should read

$$R_m = R_b = \text{if } \mathrm{C_b} > \mathrm{C_m} \text{ then } 1.0 \text{ else } 0.0 \tag{D.14}$$

- From the previous assignment, two COPS bugs were reported.

  First, changing the Mesh State Superclass had no effect. It turns out this couldn't be changed at all. This has been fixed.

  Second, there was a problem with changing the number of neighbours in an existing program. After the change was made and the code was regenerated, there were compile errors. I have not been able to reproduce this problem. If it happens to anyone, let me know.

- Here is the code for a viewer that you can use to view your results.

  To use the viewer, use the command:

```
java -jar <JAR_PATH>/BacterialDisplay.jar
     width height filename [scale]
```

  where `JAR_PATH` is the directory leading to the jar file, `width` and `height` are the width and height of the data (which is the width and height for your mesh), `filename` is the data file, and `scale` is an optional scaling factor for the display window (it defaults to a 2x2 pixel rectangle per data point).

  The window can be dismissed by simply clicking in it. If you want to grab the window, you can use the autograb feature in xv. Details are available if you wish.

  Here are two sample output files: 0 degrees and 45 degrees.

- A couple of notes on COPS I neglected to mention to one of the groups (I did mention that I always forget something, right?):

  First, you will find that attempting to remotely display the COPS GUI doesn't work well. You pretty much have to be sitting at the console to use the tool.

  Second, read over the COPS postings from the last assignment. They include information on how to run a COPS program without the GUI, submitting a COPS program, and other things you may encounter while using the tool (especially the Swing exceptions that may be thrown as you use the tool).

Finally, if you encounter any problems, contact me rather than spending time trying to work around it.

- I managed to forget to indicate the termination conditions in the assignment description.

  Add an extra parameter to the end of the parameter list for the threshold and stop when the change in concentrations for both the mould and bacteria fall below the threshold. The examples that I sent each of you used a threshold of 0.02.

  While I'm at it, your parallel version of the program should have two extra parameters for the number of horizontal and vertical partitions.

Appendix E

Mesh Computations

This appendix contains a background material on sequential mesh computations that was distributed to the subjects in the usability experiment.

E.1 Introduction

Mesh computations provide a straight–forward approach to solving a large number of problems, particularly those involving the simulation of physical surfaces or regions. Weather forecasts, particle simulations, and some image processing applications can be implemented using this approach.

This document describes the characteristics of general mesh computations, where data is represented using a general graph of nodes with connecting edges. From this general mesh, a regular mesh structure is presented. The regular mesh has the advantage of a simpler computational algorithm and structure. To demonstrate this regular structure, an example mesh application, a LaPlace equation solver, is described in detail. Finally, some alternative mesh structures are presented.

It should not be too difficult to think of questions whose answer is not addressed in this document. This is not intended to be a complete survey of this field, but rather an introduction to this style of computation.

E.2 General Mesh Computations

Simply put, a general mesh computation iterates over a set of connected data (such as the nodes in the graph in Figure E.1) computing a new value for each data element based on a combination of the current value and the values in some neighbourhood around the element.[1] The neighbourhood required to compute a new value for a given element is called its *stencil*. This iteration is

[1] In this document, the elements of the mesh refers to the nodes of the graph. In other work, the elements in a mesh may refer to the geometric regions formed by the edges of the graph.

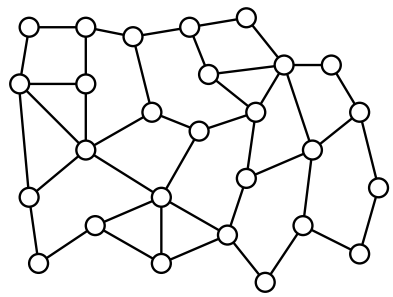


Figure E.1: An example graph for a general mesh computation.

```
Initialize graph NEW
Copy NEW to graph OLD
finished = false
while (not finished) do
    finished = true
    foreach element e in graph NEW
        Set e′ to corresponding node from graph OLD
        e.value = f(e′, neighbourhood(e′)) ;
        if (not hasFinished(e, e′)) then
            finished = false
        end if
    end for
    Copy graph NEW to OLD
end while
```

Figure E.2: The structure of a general mesh computation.

repeated until the elements satisfy the termination condition and reach their final values. The final value can be defined as the result after a fixed number of iterations, when the first element or all elements satisfy a particular condition, or the result after the data elements converge.

Pseudocode for this computation is given in Figure E.2. Note that the example code computes new values based on data from the previous iteration (we will see later that this is not always the case). This is done by using two copies of the data, one with the values from the previous iteration (`OLD`) and another to store the newly computed values (`NEW`). These graphs are then swapped before the next iteration. The code also assumes that the value from the previous iteration is needed to determine if the computation has finished, which may not be the case depending on the termination condition.

As an example, the graph in Figure E.1 could have been taken from the mesh in Figure E.3, which is used in simulating the flow of air over the wing of an airplane. The function `f()` in Figure E.2 may compute the speed and

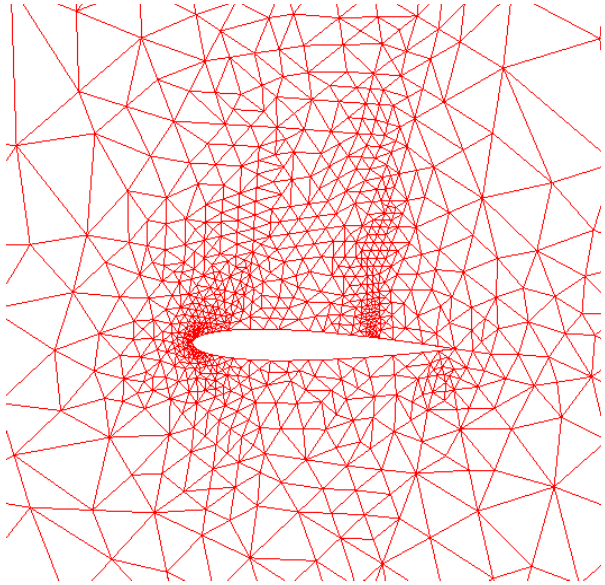


Figure E.3: A mesh for the region around an airfoil, from `http://www.cacr.caltech.edu/~roy/image/image.html`.

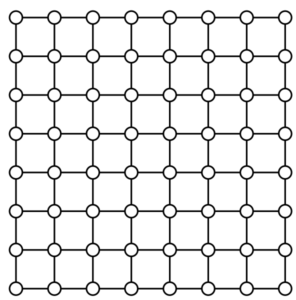


Figure E.4: An example graph for a regular mesh computation.

direction of the airflow for each mesh element based on the last calculated value and the flow in adjacent elements. The region around the cross section of the wing is decomposed into a connected graph. In a general mesh structure, this decomposition can be arbitrarily complex. For instance, in Figure E.3, the density of the graph increases in regions of particular interest, such as the area near the front of the wing. This increased density allows for finer granularity of computation in the regions that need it.

E.3 Regular Mesh Computations

In contrast to the general mesh, a regular mesh consists of a set of evenly–spaced elements, like that of Figure E.4. Otherwise, the regular mesh behaves like a general mesh.

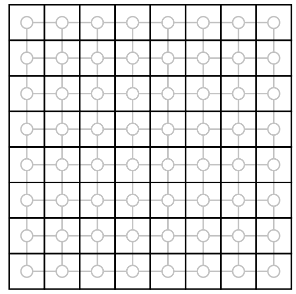


Figure E.5: Representing a regular mesh as a two–dimensional matrix.

There are two main benefits to this regular structure. First, it makes the neighbourhood around each element regular as well. Consider the function `f()` from Figure E.2. In a general mesh, this function must account for different sized neighbourhoods and change the computation accordingly. A careful examination of Figure E.3 shows that a mesh element can have between four and eight neighbours. In Figure E.4, though, there are only three cases: elements with two neighbours (at the four corners), three neighbours (at the edges, except for the corners), and four neighbours (in the interior region of the mesh). This simplifies the function for computing the new values. Second, the regular structure allows the mesh to be represented as a two–dimensional matrix, as shown in Figure E.5. The pseudocode in Figure E.2 can now be written as a simple pair of nested `for` loops.

E.4 Example: Solving the LaPlace Equation

This section provides a concrete example of a mesh computation. The example computes a solution to the LaPlace equation on a regular N by M mesh, represented by a N by M matrix. The computation is straight–forward. In each iteration, the value for each mesh element is set to the average of its immediate neighbours. This computation continues until all of the individual mesh values converge to the final answer. A mesh element has converged when the change in its value in an iteration falls below a defined threshold.

The source code for the main loop of the mesh computation is given below. This particular simulation treats the mesh elements at the four corners (indices `[0][0]`, `[0][height - 1]`, `[width - 1][0]`, and `[width - 1][height - 1]`) as constants and does not compute new values for them.[2] To keep the code simple, three loops are used to calculate new values: one loop for the top and bottom edges, one loop for the left and right edges, and one loop for the

[2]Any number of elements can be set to fixed values. The elements that have been set to fixed values in the LaPlace example were chosen to produce an interesting result without obscuring the details of the mesh computation with too much additional code to handle the fixed elements.

interior portion of the mesh. Also note that the condition for the outer loop has been changed so that the computation ends if the termination condition is not met after some maximum number of iterations. This ensures that the program always ends.

```
public void simulate()
{
    int iteration = 1 ;
    boolean converged = false ;

    while(!converged && iteration < MAXITERATIONS) {

        converged = true ;

        // Top and bottom edges
        for(int i = 1;i < this._width - 1;++i) {
            this._new[i][0] = (this._old[i + 1][0] +
                               this._old[i][1] +
                               this._old[i - 1][0]) / 3.0 ;
            this._new[i][this._height - 1] =
                              (this._old[i + 1][this._height - 1] +
                               this._old[i][this._height - 2] +
                               this._old[i - 1][this._height - 1]) / 3.0 ;

            if (Math.abs(this._new[i][0] - this._old[i][0]) > THRESHOLD ||
                Math.abs(this._new[i][this._height - 1] -
                         this._old[i][this._height - 1]) > THRESHOLD) {
                converged = false ;
            } /* if */
        } /* for */

        // Left and right edges
        for(int j = 1;j < this._height - 1;++j) {
            this._new[0][j] = (this._old[0][j - 1] + this._old[0][j + 1] +
                               this._old[1][j]) / 3.0 ;
            this._new[this._width - 1][j] =
                              (this._old[this._width - 2][j] +
                               this._old[this._width - 1][j - 1] +
                               this._old[this._width - 1][j + 1]) / 3.0 ;

            if (Math.abs(this._new[0][j] - this._old[0][j]) > THRESHOLD ||
                Math.abs(this._new[this._width - 1][j] -
                         this._old[this._width - 1][j]) > THRESHOLD) {
                converged = false ;
            } /* if */
        } /* for */

        // Interior
        for(int i = 1;i < this._width - 1;++i) {
            for(int j = 1;j < this._height - 1;++j) {
                this._new[i][j] = (this._old[i + 1][j] +
                                   this._old[i - 1][j] +
                                   this._old[i][j + 1] +
                                   this._old[i][j - 1]) / 4.0 ;

                if (Math.abs(this._new[i][j] - this._old[i][j]) > THRESHOLD) {
                    converged = false ;
                } /* if */
            } /* for */
```

```
        } /* for */

        // Swap references to old and new arrays
        this.swapArrays() ;
        ++iteration ;
        System.out.println("Iteration " + iteration) ;
    } /* while */
} /* simulate */
```

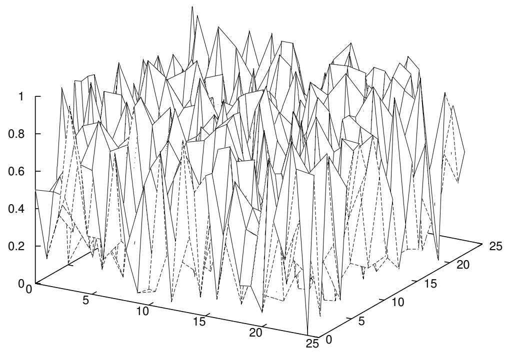
An example of the results from the LaPlace solver are shown in Figure E.6, for a 25 by 25 element mesh. In this case, the elements at indices `[0][0]` and `[width - 1][height - 1]` are both set to 0.5, the element at `[0][height - 1]` is 1.0, and the element at `[width - 1][0]` is 0.0. The initial values for the other elements are set to random numbers between 0 and 1, shown in Figure E.6(a). In Figure E.6(b) the solution is beginning to take shape after five iterations. Figure E.6(c) shows the final solution of the LaPlace solver with the threshold set to 0.025, which required 138 iterations of the mesh loop. Although this solution would be considered accurate (the threshold yields a 2.5% margin of error), the surface does not appear smooth. Lowering the threshold to 0.001, for a 0.1% margin of error, produces the much smoother surface in Figure E.6(d). However, this solution takes 2685 iterations to converge, over 19 times more than the less accurate solution.[3] The need for an accurate solution must be weighed against the additional time needed to compute it.

E.5 Alternative Mesh Structures

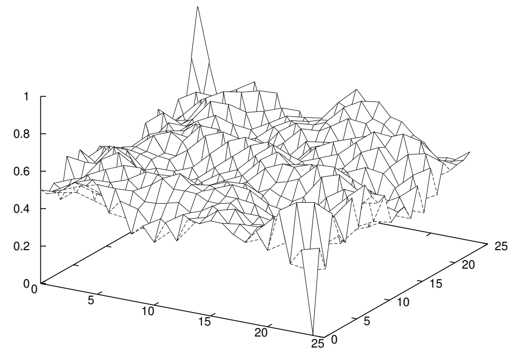
The LaPlace example in the previous section is one of the most basic mesh computations. There are many different alternatives that can be used to address the needs of more specific problems. This section discusses some of these alternatives.

The first alternative structure deals with the *topology* of the mesh, or how mesh elements on the boundary are handled. The LaPlace solver has a *non–toroidal* topology, where the edges represent the end of the data. There are other ways of handling the edges of the data, shown for two dimensions in Figure E.7. These topologies can be used to simulate different–shaped surfaces for two–dimensional data. For instance, a horizontal–toroidal or vertical–toroidal mesh forms a cylindrical surface. A fully–toroidal mesh creates a torus (generally referred to as a donut), shown in Figure E.8. However, it is not necessary to treat the surface this way. It is straight–forward to unroll both the cylinder and torus back to two–dimensional data. Further, it is possible to extend these topologies to meshes with three or more dimensions.

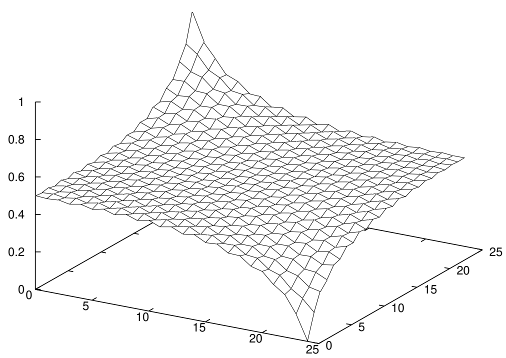
[3] In this particular case, the increase in the number of iterations appears to scale up at approximately the same rate as the change in accuracy. However, this is a coincidence. Setting the threshold to 0.01 takes 629 iterations to converge, a 4.5–fold increase for a 2.5–fold increase in accuracy.



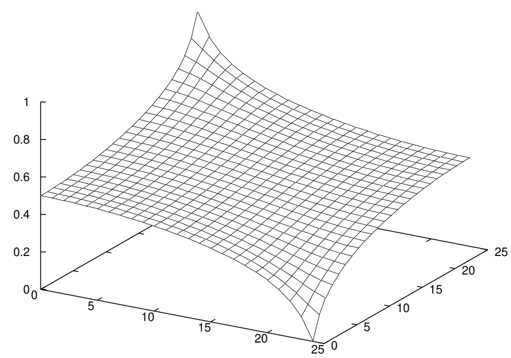
(a) The starting conditions, with random initial values.



(b) After 5 iterations.

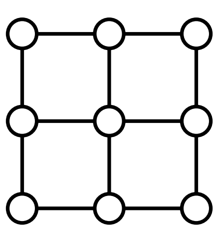


(c) Converged solution with a threshold of 0.025 (138 iterations).

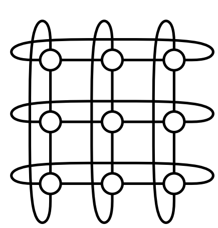


(d) Converged solution with a threshold of 0.001 (2685 iterations).

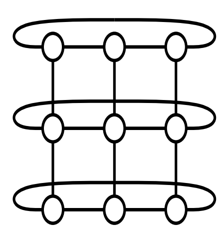
Figure E.6: The LaPlace equation solver at various stages of completion.



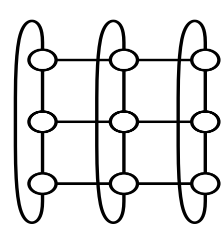
(a) A non–toroidal mesh.



(b) A fully–toroidal mesh.



(c) A horizontal–toroidal mesh.



(d) A vertical–toroidal mesh.

Figure E.7: Different topology options for a mesh.

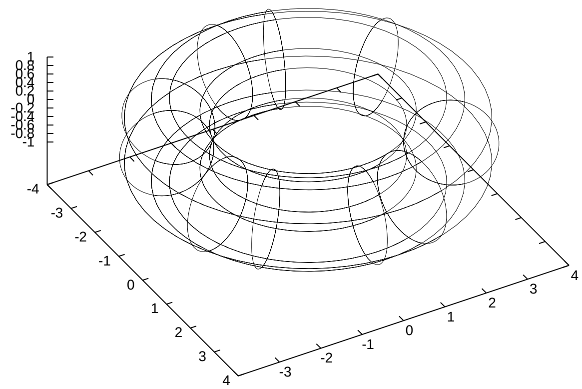


Figure E.8: A torus, which can be simulated using a folly–toroidal mesh.

(a) An eight–point mesh.

(b) An arbitrary stencil using 16 neighbours.

Figure E.9: Different stencils for a mesh computation, used to compute a new value for the grey element in the centre.

Another useful mesh structure involves using different stencils to compute new values. The LaPlace solver uses a *four–point* stencil (also called a four–point mesh), using the four neighbours on the four compass points to calculate the new value for each element. Alternately, an *eight–point* mesh, shown in Figure E.9(a), can be used. More general stencils, such as that in Figure E.9(b), may be required for some problems.

As noted earlier, there are several different possibilities with respect to determining when a mesh computation has finished. Among the various options, a computation may be finished when

- the data values converge. There are 2 possibilities for checking convergence:

    1. *Local termination conditions.* This condition checks that the change in value in each individual mesh element falls below a threshold. This was used in the LaPlace solver.
    2. *Global termination conditions.* This condition checks that the sum of the changes over the entire mesh falls below a threshold.

- one (or more) elements satisfy a given condition. If the application is simulating the flow of heat through a material, the simulation may halt if the melting point of the material is ever reached.

- a maximum number of iterations has elapsed. This condition may be used in conjunction with any other termination to ensure that a computation eventually halts.

Another issue in a mesh computation is the style of iteration, or rather which data is used to calculate new values in each iteration. In the example and pseudocode earlier, new values are computed based on data from the previous iteration, which is called *Jacobi iteration*. The alternative is *Gauss–Seidel iteration*, sometimes called *over–relaxation*, which uses results calculated earlier in the same iteration to calculate new values. For example, rewriting the body of the third loop of the LaPlace solver using Gauss–Seidel iteration (lines 46 through 49) yields

```
this._new[i][j] = (this._new[i + 1][j] + this._new[i - 1][j] +
                   this._new[i][j + 1] + this._new[i][j - 1]) / 4.0 ;
```

Both iteration styles have advantages and disadvantages. The advantage of Jacobi iteration is:

1. The results are not dependent on the order in which the mesh elements are processed.

The disadvantage of Jacobi iteration is:

1. It always requires additional memory for an extra copy of the data. The amount of extra memory can sometimes be optimized by having each element keep two copies of the data (a read copy, the equivalent of `old` from the LaPlace solver, and a write copy, the equivalent of `new`). Like the copying step in Figure E.2, though, this update requires an iteration over the mesh elements, but this optimization allows larger problems to be solved. On the other hand, if a complete second copy of the mesh data can be kept then a program using Jacobi iteration can swap the two data sets. This swap may only require the exchange of the two references to the two data sets, which is faster than either copying the data or updating each element. The needs of the application will determine if it is more important to optimize memory use to solve larger problems or reduce the execution time by swapping the two mesh data sets.

The advantages of Gauss–Seidel iteration are:

1. It can use less memory. If the program does not terminate on convergence, then no extra copy of the data is needed. If the extra copy of the data is needed, then the memory saving idea from Jacobi iteration may be used. However, it is not possible to swap the two data sets as this leads to incorrect results.

2. It is sometimes necessary in order to ensure that a problem converges. In fact, it is possible that the LaPlace solver will fail to converge on some problems unless Gauss–Seidel iteration is used.

The disadvantage of Gauss–Seidel iteration is:

1. The solution is sensitive to the order in which the elements are evaluated. Any changes to the processing loops will be reflected in the final answer. Further, if the method of enumerating over the mesh elements does not guarantee an ordering, the results may be non–deterministic. However, the solution produced will be correct within the bounds of the termination condition. This only presents a problem if the program must yield consistent results.

E.6 Conclusion

This document discussed some of the basics of mesh computations, concentrating on regular meshes and some of the options available.

Appendix F

Parallel Mesh Computations

This appendix contains a background material on parallel mesh computations that was distributed to the subjects in the usability experiment.

F.1 Introduction

Mesh computations provide a straight–forward approach to solving a large number of problems, particularly those involving the simulation of physical surfaces or regions. Weather forecasts, particle simulations, and some image processing applications can be implemented using this approach.

This document is an addendum to "Mesh Computations." It describes a basic strategy for parallelizing regular mesh computations based on spatial decomposition of the mesh elements. Some of the issues that arise in parallel mesh computations are also discussed.

This document is not intended to discuss all of the issues and variations of parallel mesh computations. However, the information should be enough to write a simple parallel mesh program.

F.2 Parallelization Strategy

A mesh computation is usually parallelized by decomposing the mesh elements into a disjoint set of partitions. Each of these partitions is assigned to a processor[1] that is responsible for executing the mesh computation for the elements assigned to it. Of course, each mesh element uses the values from neighbouring elements, so the processors must exchange their boundaries to compute new values for the elements on the edge of their assigned partition.

Pseudocode for the complete parallel mesh computation is given in Figure F.1. The code for an individual processor is shown in Figure F.2. Note that this structure is a general parallel mesh program, so not all steps may be

[1] In this document, a processor refers to a process or thread that may or may not be executing on a separate CPU.

```
Initialize mesh data M
foreach processor p
    Determine M_p, the partition of M to be assigned
        to processor p
    Start the execution on processor p with M_p
        (See Figure F.2)
end for
foreach processor p
    Wait for p to finish
    Get result mesh data R_p from p
end for
foreach result R_p
    Gather R_p into final result R
end for
```

Figure F.1: The overall structure of a parallel mesh computation.

necessary. Further, the pseudocode is independent of issues such as iteration style and synchronization. Many of these additional issues are extensions of the issues raised in a sequential mesh computation, and are discussed in more detail later in this document.

From Figures F.1 and F.2, there are four issues in a parallel mesh program:

1. Partitioning the mesh data for the processors,
2. Exchanging the boundary,
3. Evaluating the termination conditions, and
4. Gathering the final results.

These issues may be affected by the differences between Jacobi iteration versus Gauss–Seidel iteration and distributed memory computers versus shared memory computers. The following four sections discusses each issue in more detail.

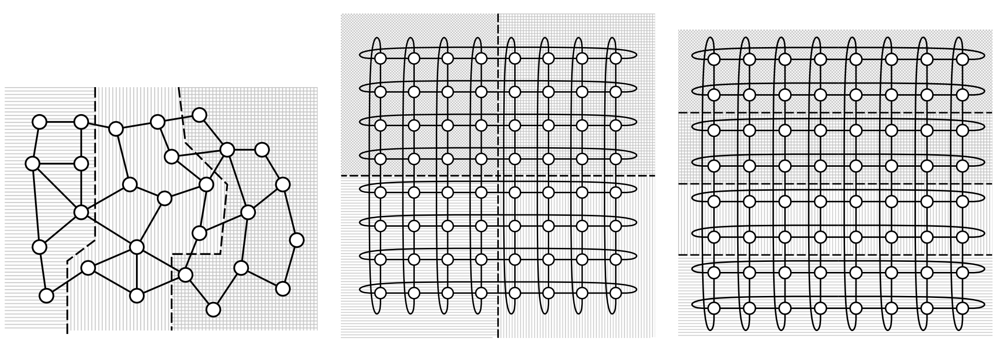
F.3 Partitioning the Mesh Data

The first issue in a parallel mesh computation is deciding on how to partition the mesh data so that it can be distributed across the processors. Typically, the mesh data is spatially decomposed, as shown by the three examples in Figure F.3. Normally, the partitions are a contiguous set of connected elements, but this need not be the case.

When partitioning the mesh data, there are two competing forces that must be simultaneously balanced:

```
Perform parallel preprocessing of M_p
while (notDone()) do
    foreach element in local partition M_p
        Preprocess element
    end for
    Exchange boundary with adjacent partitions
    foreach element in local partition M_p
        Compute new value for element
    end for
end while
Perform parallel postprocessing of M_p
```

Figure F.2: The structure of a parallel mesh computation for an individual processor.



(a) A general mesh broken into three partitions.

(b) A regular mesh broken into four rectangular partitions.

(c) A regular mesh broken into four striped partitions.

Figure F.3: Examples of partitioned meshes for a parallel implementation of a mesh computation.

1. The amount of computation for each partition should be the same, so that all of the processors finish at about the same time and none are idle for an extended period. Balancing the load over the available processors allows for more effective use of the available processors and thus yields faster results.

2. The amount of communication required to exchange the boundaries should be minimized. Communication is an overhead in a parallel program – it is extra work that does not need to be done in a sequential program.

Determining the ideal partitioning of mesh data, particularly for a general mesh, is the subject of current research and generally requires the use of other tools. Some research has been devoted to adjusting the partitions as the computation progresses. For a regular mesh, though, a rectangular partitioning (Figure F.3(b)) or striped partitioning (Figure F.3(c)) is usually best.

F.4 Exchanging the Boundaries

When calculating the new value for an element, a mesh computation uses the values from neighbouring elements. To parallelize such a computation, though, the mesh data is spatially decomposed into a set of partitions that are distributed over processors. Now, for elements on the edge of a partition, some of these neighbouring elements may have been assigned to another processor and may not be available locally. To compute new values for these elements, each processor must now perform a *boundary exchange* with those processors that have mesh elements needed to complete the calculation of the local partition. This boundary exchange defines the communication structure of the program and is the most crucial synchronization point in a parallel mesh computation. This section discusses this exchange and some of the issues involved.

The precise nature of this exchange depends on three factors: the topology of the mesh, the memory architecture of the parallel machine, and the type of iteration used in the program.

The topology of the mesh affects which processors are considered to be neighbours, and this affects the communication structure of the program. This factor is analogous to requiring that the function for computing new values for a mesh element be able to deal with different sized neighbourhoods, depending on the location of the element and the topology.

There are two basic memory architectures used in parallel machines. In a distributed memory machine, each processor has a separate physical memory that only it can access. Other processors must send network messages to get copies of data. In such a machine, the boundary exchange uses a "ghost boundary," a set of elements that hold a copy of the elements on the edge of the adjacent partition. It is important to note that the ghost boundary is not part of the partition that is assigned to a processor, but is a copy of data that

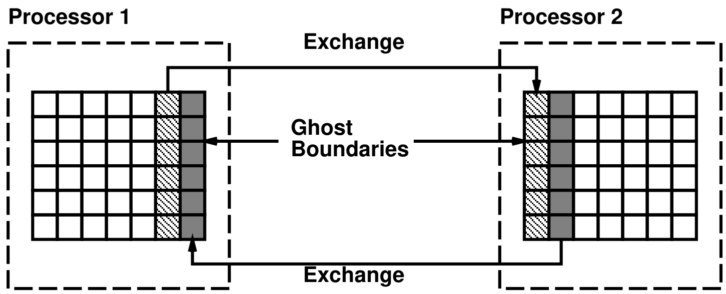


Figure F.4: An example of exchanging boundaries using a ghost boundary. The ghost boundaries are the right column in Processor 1 and the left column in Processor 2.

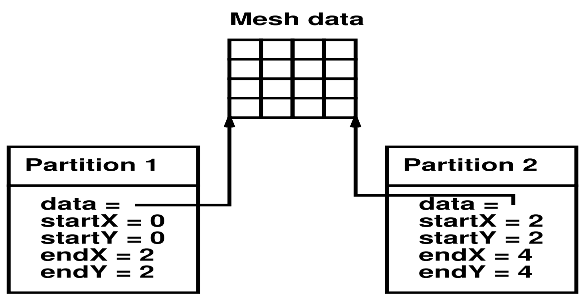


Figure F.5: Logically partitioning mesh data, by referring to a single copy of the data but adding wrapper objects to set the bounds owned by a processor. `Partition 1` refers to the upper left quadrant, and `Partition 2` refers to the lower right quadrant.

is used to compute new values for elements on the edge of the local partition. An example of an exchange using the ghost boundary is shown in Figure F.4.

A shared memory machine can also use a ghost boundary. However, to reduce copying overhead, each processor may simply be able to access the mesh elements it needs, depending on the type of iteration used. In this case, it is possible to logically partition the mesh data, as shown in Figure F.5. A partition can access elements in neighbouring partitions by simply accessing the shared mesh data. For Jacobi iteration, this requires that each element have both its old value (which is read by other elements to compute new values) and a new value that has been calculated during the current iteration. See the document "Mesh Computations" for more information on the issues related to having each element hold both of these values. Since the boundaries are not copied, there is no actual exchange. However, synchronization is still necessary, as explained below.

However, the choice between Jacobi and Gauss–Seidel iteration has the greatest effect on the boundary exchange. This choice affects both how often the boundaries are exchanged and the amount of synchronization needed in the program. Before these issues are discussed, it is necessary to note the differences in the semantics in the two iteration styles when applied to a parallel program.

As in the sequential case, Jacobi iteration always computes new values for mesh elements based on the results from the previous iteration. These semantics require that

- the boundary exchange for a given iteration be completed before a processor starts computing new values for its local partition, and
- the iterations be done in lock step in all threads. This simply means that no processor should start the next iteration before all other processors have completed the previous one.

To ensure that these semantics are met, it is necessary to introduce synchronization into the boundary exchange and at other points in the mesh computation (these will be addressed later). This synchronization consists of a *barrier* after the exchange in Figure F.2. A barrier is a synchronization structure that all processors must enter before any can leave. This ensures that no processor can compute new values using old neighbour data as the boundary exchange must be completed first.

Gauss–Seidel iteration has a new meaning in a parallel program, though. In a sequential program, Gauss–Seidel iteration computes values for the mesh elements using values calculated earlier in the iteration. For a local partition, this is still the case. In a parallel program, Gauss–Seidel iteration also relaxes the requirements that each iteration be executed in lock step and that the boundary exchange must be done in each iteration. This is done by removing the barrier synchronization and modifying the exchange to poll for new data from neighbouring partitions. If new data is available, the next iteration uses these new values. If not, then the next iteration uses the old values. In a shared memory machine that simply accesses the mesh elements in neighbouring partitions, Gauss–Seidel iteration simply uses the most recent values that are available.

As in the sequential case, both Jacobi and Gauss–Seidel iteration have advantages and disadvantages when applied to a parallel program. The advantage of Jacobi iteration is:

1. If implemented properly, both the sequential and parallel versions of the same program will produce identical results. This makes it easy to verify that the parallel version is working correctly.

The disadvantage of Jacobi iteration is:

1. Parallel programs may take longer to finish than those written using Gauss–Seidel iteration. The barrier synchronization in Jacobi iteration adds overhead to a parallel program.

The advantage of Gauss–Seidel iteration is:

1. Parallel programs may finish faster because of the reduced synchronization.

The disadvantage of Gauss–Seidel iteration is:

1. Parallel programs based on Gauss–Seidel iteration will produce non–deterministic results. The non–determinism arises because an iteration may or may not use current data from neighbouring partitions, depending on when this data becomes available. Further, there are issues regarding the correct detection of convergence when using Gauss–Seidel iteration. These are discussed in Section F.5.

F.5 Evaluating the Termination Conditions

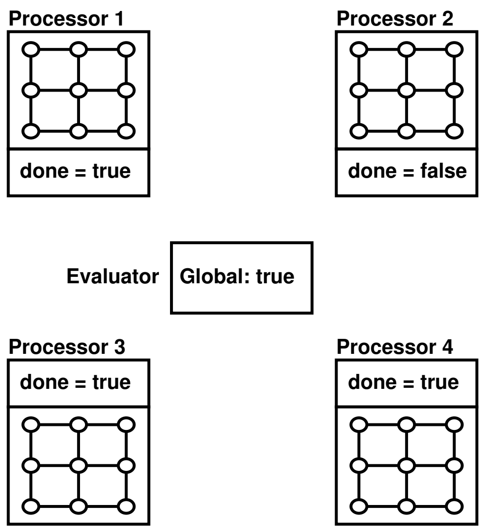
An important part of every program is determining when it has finished. In a parallel program, this can be a difficult problem as it normally requires some communication between processors and some co–ordination to ensure that the program has actually finished. This section first gives a basic strategy for evaluating the termination conditions in a parallel mesh program. This strategy is for problems with local termination conditions, but it should be clear how to extend it to deal with global conditions. However, this is complicated by the slightly different semantics of a Gauss–Seidel iteration in a parallel program.

A simple strategy for evaluating termination conditions is shown in Figure F.6. This strategy uses a small region of shared memory. Note that since local termination conditions are used, the only information that need be sent between processors is a Boolean value that indicates whether all of the mesh elements in the local partition have finished. This strategy is a possible implementation of the `notDone()` call in Figure F.2.

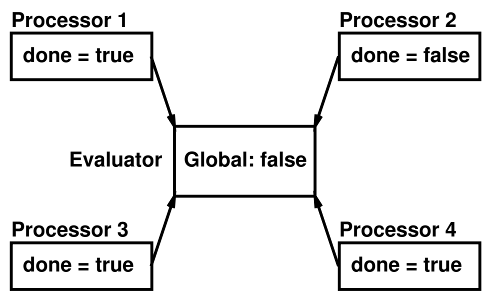
The main issue with respect to termination conditions is the style of iteration used in the program. This issue affects the synchronization that is needed when evaluating the condition and the reliability of the check.

In a program using Jacobi iteration, the complication is the synchronization needed to ensure correct results. Both the iterations of the computation and the evaluation of the termination conditions need to be done in lock step. In the strategy from Figure F.6, three barriers must be used, after the steps in Figures F.6(b), F.6(c), and F.6(d). The first barrier ensures that each processor sends its local flag to the Evaluator before any processor tries to access the global result. The synchronization in the method that gathers these flags is required to ensure that the global flag is set properly. The second barrier ensures that the global flag is not reset before each processor has had a chance to obtain the value. Finally, the last barrier ensures that no processor tries to start evaluating the termination conditions (by sending its local flag in Figure F.6(b)) before the global flag has been reset.

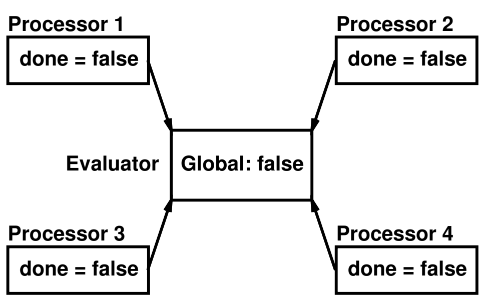
In a program using Gauss–Seidel iteration, the complication is in properly determining if the mesh data in a program has converged. Like the compu-



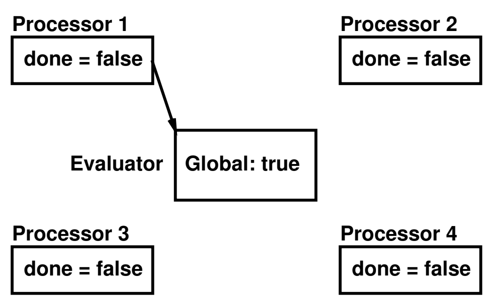
(a) Each processor determines if all of its elements have finished and sets a local flag. Initially, the global result in the Evaluator object is true.



(b) Each processor sends its flag to the Evaluator (via a synchronized method call). As the Evaluator receives these flags, it sets the global result.



(c) Each processor reads the global flag from the Evaluator (via an accessor method). The result is used to decide if the computation should continue.



(d) A designated processor resets the global result for the next evaluation of the termination conditions.

Figure F.6: A simple strategy for evaluating the termination conditions using shared memory.

tation, no synchronization is used when checking for convergence. Thus, the data used in the check for convergence may be old and the program may incorrectly finish. Normally, additional sequential iterations must be done after the parallel code has finished to verify that the data has truly converged. These extra iterations may outweigh the savings from the reduced synchronization. Note, though, that this only happens in programs that converge to their final answers.

F.6 Gathering the Final Results

The last step in a mesh computation is to gather the final results. This is particularly true in distributed memory systems, where the mesh data is copied to different physical memories. The final answer must be assembled on a designated processor. However, it may also be required in a shared memory environment if the mesh data is partitioned by copying it rather than logically partitioning it.

In some mesh computations, the final answer is not the new mesh data values. Instead, the mesh data may be *reduced* to a single value. For instance, if the program is testing the airflow around the wing of an airplane, the final answer may be a Boolean indicating if the wing generates sufficient lift for the plane. Under certain conditions, this reduction can be done in parallel. However, parallel reductions are beyond the scope of this document.

F.7 Conclusion

This document described a basic approach to parallelizing a mesh computation. Further, some of the issues that arise in the parallel implementation were also discussed.

Appendix G

Choice Points Used for the Usability Experiment

Choice points are a measure of the complexity of a program. Choice points are those points of a program that can alter the flow of control and may no longer be sequential, such as selection control statements or loops. Errors in programs tend to occur at these points when the programmer makes the wrong choice of what to do next. For example, programmers tend to make mistakes when checking the boundaries of arrays (and then treating those elements incorrectly) rather than at the interior of the data where no checks are needed.

Note that choice points are distinct from the size of a program, although not completely independent (a larger program will tend to have more choice points than a smaller program). However, choice points are a better approximation of the complexity of a program. For practical purposes, a long program that consists of a sequence of statements is simpler than a small program with many loops and control statements. This is in contrast to the idea that the size of the program determines its complexity. One could even go further and suggest *choice point density*, dividing the number of choice points by the size of the program. A higher density suggests more complex code.

The choice points used for the usability experiment are the following:

`?`, `if`, `else`, and `else if` : The first three are counted, and the fourth is subtracted from the total of choice points.

&& and `||` : Each clause in a boolean operation is a different choice point and should be counted. Each clause can influence the flow of control.

`case` and `default` : These two statements count the clauses in a `switch` statement.

`for` and `while` : These two statements catch all of the loops in a program. The `while` will count both `while` and `do` loops.

`catch` : This captures the changes in control flow from exceptions. The `try` statement is not used because it does not alter the flow of control, but rather delineates a basic block in which exceptions may occur. Further, there can be multiple `catch` statements for a given `try` statement.