University of Alberta

Library Release Form

Name of Author: John Karsten Anvik

Title of Thesis: Asserting the Utility of CO_2P_3S Using the Cowichan Problems

Degree: Master of Science

Year this Degree Granted: 2003

John Karsten Anvik
706 Lakewood Rd N,
Edmonton, Alberta
Canada, T6K 3Z8

Date: ____________________

University of Alberta

Asserting the Utility of CO_2P_3S Using the Cowichan Problems

by

John Karsten Anvik

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

Department of Computing Science

Edmonton, Alberta
Fall 2003

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 **Asserting the Utility of CO_2P_3S Using the Cowichan Problems** submitted by John Karsten Anvik in partial fulfillment of the requirements for the degree of **Master of Science**.

Duane Szafron
Supervisor

Jonathan Schaeffer
Supervisor

Eleni Stroulia

Mike Carbonaro

Date:

In memory of Buzz, Molly, Rainbow, and Charlotte.

Abstract

Parallel programming is seen as an effective technique to improve the performance of computationally-intensive programs. This is done at the cost of increasing the complexity of the program, since new issues must be addressed for a concurrent application. Parallel programming environments provide a way for users to reap the benefits of concurrent programming while minimizing the effort required to create them. The CO_2P_3S parallel programming system is one such tool which uses a pattern-based approach to create parallel programs.

This dissertation demonstrates that the CO_2P_3S system contains a sufficient number of parallel patterns to implement a wide variety of applications. This characteristic is called the *utility* of a system. The Cowichan Problem set is a collection of problems for assessing how easy a parallel programming environment is to use (called the system's *usability*), and is used here to show the utility of CO_2P_3S.

Each of the applications from the Cowichan Problems is presented along with the pattern used to implement a solution. If an application could not be created because a pattern was lacking, then the necessary pattern was added to CO_2P_3S. This demonstrates the *extensibility* of CO_2P_3S . As a result of this work, two new patterns were added to CO_2P_3S: the Wavefront pattern and the Search-Tree pattern. Code metrics and performance results are presented for the various applications in order to show that the use of the CO_2P_3S system can greatly reduce the effort required to create concurrent programs that have reasonable scalability.

Acknowledgements

First, I wish to acknowledge the efforts of my two supervisors, Duane Szafron and Jonathan Schaeffer, who guided the development of this work. Their help was invaluable at all stages of its creation, most especially in the designing of the new patterns and the organization of this dissertation.

Also, this work could not have been done without the creation of the CO_2P_3S system by Steve MacDonald and the development of MetaCO_2P_3S by Steve Bromling. Steve MacDonald was especially invaluable for his assistance with respect to the technical details of developing new patterns, the intricacies of concurrent programming in Java, and the writing of a dissertation. Thanks also to Kai Tan, the other member of the CO_2P_3S research group, who helped in the selection of this topic and its development there after.

I thank the members of my committee for their efforts on my behalf. Their comments regarding this work were an important part of its completion.

Also deserving of recognition are (in no particular order) Cam Macdonell, Maria Cutumisu, Ernie Novillo, James Redford, and Steve MacDonald who provided the friendly environment in which this research grew. Much of the humour and laughter needed to lighten a frustrating day were found through them.

Finally, I must acknowledge the efforts of my wife Danielle who gives me the freedom to explore my interests to my heart's content. She took care of life's many details so that I could focus on this research. With insufficient background to understand most of the material in this work, she read it in its entirety and made many helpful comments. In her own way she produced this work also. My daughter Tara also needs to be acknowledged, as she provided in play much of the distraction necessary to complete this work in its latter stages. My twin boys, John and Erik, also deserve recognition, as their arrival provided the last little bit of motivation necessary to complete this work.

If I have forgotten anyone, my apologies. It is not so much insensitivity on my part as forgetfulness of old age at 28.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In many fields of research there exist problems which simply take too long to find an answer to using a single computer. Given an algorithm to determine the weather next week, it is useless if it takes ten days to compute the result. While upgrading the processor to a faster model or increasing the amount of available memory can improve the computational time, possibly reducing the ten days by a few days, these problems may still be too large to find the solution in a reasonable amount of time. Significant improvement can only be made by dividing the problem into independent portions and using multiple processors to simultaneously compute them. Assuming that the computation of next week's weather can be evenly split in half, two processors could return next week's forecast in five days, allowing enough time to put the snow tires on your car or to cancel you weekend picnic plans.

However, this benefit is not without cost. The cost of improving the computational time for a problem in this manner is an increased complexity of the algorithm, or the use of a completely different algorithm. Adding parallelism to an algorithm adds new concerns such as synchronization and communication between the processors. It also makes debugging programs more difficult as non-determinism is introduced. In general, writing parallel programs is a complex and error-prone task, even for experts in the field.

However, there is hope. In sequential programming, there exist strategies which may be used across many problems. These strategies are called *design patterns* and they encapsulate the knowledge of solutions for a class of problems. To solve a problem using a design pattern, the pattern need only be chosen and adapted to that particular problem. By referring to a problem by the particular strategy that may be used to solve it, a deeper understanding of the solution to the problem is conveyed and certain design decisions are implicitly made. If a problem is described as using a "stack", there is an immediate understanding of some of the behaviour of the solution. There is also an implicit understanding of design decisions and the consequences of those decisions.

Design patterns do not represent a single layer of design abstraction, in that "one person's pattern is another person's primitive building block" [12]. Abstract data structures such as stacks and trees may be viewed as design patterns at a certain level of abstraction, as could complex, domain-specific frameworks.

One of the problems with design patterns is that in general they give a description of the solution but not the solution itself. Design patterns describe a solution technique in words, not in code. Once a design pattern has been chosen, the programmer must then implement the pattern and adapt it to the specific application. The amount of effort required to adapt the design pattern may range from trivial to substantial. Consequently, many tools have been created to minimize the effort required by the programmer in implementing a specific design pattern. Budinsky *et al.* [8] created a Web-based tool for generating the code for the "Gang of Four" design patterns [12], where the patterns are parameterized. The PSiGene tool [25] takes the opposite approach and narrows the application domain to only patterns for generating building simulators, such as a simulator for a room. This narrowing of the application domain removes the variations in the pattern design. The user selects specific instances of the necessary patterns for their application. This results in many specialized implementations of the design patterns, and the user simply selects the appropriate one to generate the necessary simulator. Commercial pattern-based code generation tools also exist such as Together ControlCenter [9] and ModelMaker [29].

Frameworks are at the other end of the design spectrum from design patterns. Whereas a design pattern is an abstract description of solution, a framework is a concrete implementation of a portion of a solution. A *framework* provides the application-independent structural code for a particular solution, typically through a collection of abstract classes for a specific problem domain. One domain in which frameworks are used extensively is graphical user interfaces. A framework is used by implementing *hook methods* which contain application-specific code, such as the function of a button. These hook methods are then called in the application though the framework. Frameworks provide a maximum of code reuse, unlike design patterns which must be re-implemented each time that they are used. However this re-usability comes at the cost of less generality.

Just as there are sequential design patterns, there exist *parallel design patterns* which capture the synchronization and communication structure for a particular parallel solution. Examples of common parallel design patterns are the fork/join model, pipelines, meshes, and work piles. The design patterns described in this work are at this level of abstraction, that of communication and synchronization of pieces of work between multiple processes.

The CO_2P_3S[1] parallel programming system eases the effort required to write parallel programs through its use of *adaptive generative parallel design patterns*. An *adaptive design pattern* is an augmented design pattern which is parameterized so that it can be readily adapted for an application. Budinsky's tool [8] is similar to CO_2P_3S in that it also uses parameterized patterns. The CO_2P_3S system lets a user select a parallel design pattern, adapt it for an application through providing values for the parameters, and then generate the structural code for that portion of the application. In this way, CO_2P_3S uses a design pattern to generate a framework which then becomes a parallel program.

One of the limitations of Budinsky's tool is how the patterns are included into the user's code. The user is required to cut-and-paste the generated code from a web-browser into a source code file. As the authors admit, it may not always be obvious what to cut out and where to paste it. Another limitation of the tool is that once the generated code has been integrated into an application, any regeneration of the pattern code due to a changing of pattern parameters will require reintegration of the generated code into the application. Since CO_2P_3S generates Java source files, the user is never concerned about where the generated code is to be placed in the application or about having to reintegrate code if parameters change; both occur automatically.

CO_2P_3S is an example of a parallel programming system. Many parallel programming systems have been constructed to aid in the creation of parallel programs. Some systems are designed to create parallel programs for a very specific domain, while others produce more generalized parallel programs. Some, such as PAS [13][14] and Enterprise [24], are pattern-based like CO_2P_3S. Given the number of different parallel programming systems which have been written, work has been done to determine what attributes characterize a good parallel programming system. An ideal set of characteristics for a parallel programming system was proposed in Singh *et al.* [26]. One of the characteristics which is obviously necessary for a parallel programming system is *usability*, the ease with which a system can be learned and used. Another important characteristic for a pattern-based programming system is the *utility* of the system, or that it can be used across a wide range of applications.

Several studies have been done to assess the usability of various parallel programming systems. There has even been an attempt to create a benchmark problem set for testing the usability of parallel programming systems, called the *Cowichan Problem Set* [31]. While the usability of a programming system is certainly of great importance, the utility of a system

[1] Correct Object-Oriented Pattern-based Parallel Programming System, pronounced 'cops'.

is often glossed over, despite the observation that this is a major flaw in many parallel programming systems.

This dissertation demonstrates the utility, usability, and extensibility of the CO_2P_3S parallel programming system. Utility is demonstrated by showing the breadth of applications which can be written using the tool. As the Cowichan Problem set covers a wide range of applications, it is used to show the utility of the system. The usability of the CO_2P_3S system is also shown via the implementation of the problem set. A description of how each of these problems was implemented using the patterns existing in CO_2P_3S is given, and where a specific pattern was lacking, how a new pattern was added to the system. This will demonstrate the extensibility of CO_2P_3S.

A major contribution of this dissertation is the addition of two new patterns to the CO_2P_3S system: the Wavefront pattern and the Search-Tree pattern. Also, the Search-Tree pattern introduces the notion of verification parameters to the CO_2P_3S system. The utility, usability, and extensibility of the CO_2P_3S system are demonstrated through the implementation of a variety of applications.

A description of some of the previous pattern-based parallel programming tools that have been constructed are presented in Chapter 2. Also presented in this chapter is some of the previous work done on assessing the usability of parallel programming systems such as CO_2P_3S , and an overview of the Cowichan Problems. CO_2P_3S and MetaCO_2P_3S (the tool for adding patterns to CO_2P_3S) are then described in Chapter 3. Each of the next four chapters describes the four patterns which were used to implement the Cowichan Problems and how the patterns solved the problems. Finally, future work and some concluding remarks are made in Chapter 8.

Chapter 2 provides the background for some of the previous pattern-based parallel programming tools that have been constructed, previous work on assessing the usability of such systems, and an overview of the Cowichan Problems. CO_2P_3S and MetaCO_2P_3S (the tool for adding patterns to CO_2P_3S) are then described in Chapter 3. Each of the next four chapters describes the four patterns which were used to implement the Cowichan Problems and how the patterns solved the problems. Finally, future work and some concluding remarks are made in Chapter 8.

Chapter 2

Background

2.1 Parallel Pattern-based Programming

Work with respect to patterns in parallel programming has been going on for more than ten years. An early example of finding patterns in parallel programming is the Munin project, which observed for parallel programs that "a large percentage of shared data accesses fall into a relatively small number of access type categories" [3]. Nine unique shared memory data object types were identified and used to annotate shared program variables in order to indicate the variable's memory access pattern. The Munin system then used these annotations as a guide to the memory consistency protocol to use for a particular shared variable. These patterns represent a very low level of abstraction, the level of memory consistency.

Parallel Architectural Skeletons (PAS) [13][14] is a C++ library which encapsulates the structure and behaviour of a parallel design pattern in an application-independent manner. A PAS user extends a *pattern skeleton* by providing the values for various structural and behavioural pattern parameters to create an *abstract parallel computing module*, which is then refined by adding application-specific code to form a *concrete parallel computing module*. The parallel application is a collection of interacting concrete parallel computing modules. The various skeletons in PAS represent patterns for pipeline computation, divide-and-conquer parallelism, data-parallel mesh-style computation, and dynamic replication.

Another C++ library is TACO (Topologies and Collections) [22]. TACO provides C++ templates which may be used to create *object groups* which are distributed across a network and upon which *collective* methods can be used to do the computation in parallel. The user defines *functors*, functions that inherently contain state, which are then used by the collective operations to perform the computation. Creating a TACO application involves specifying the necessary functors, specifying the topology of the application using TACO *object groups*, and sending the appropriate message to the *leader* of the object group. The topology is constructed by creating an object group leader and then adding objects to the group via the group leader. Synchronization is also accomplished through messages to the group leader. Sending a message to the leader of an object group is similar to sending a message to the root of a tree and having the operation propagate through the tree.

The Tracs [2] system allows users to develop parallel applications by graphically specifying the communication between processes. While a user could create arbitrary parallel application structures, the system provides certain pre-defined "architecture models" which the user can use either as a description of the specific application or as a starting point for describing the parallel structure of the application. Also, nodes of the architecture graph can be assigned pre-defined functions such as a filter, master, worker, or pipeline element. Once the structure of the application is defined, including specifying the function of certain elements, the application code can then be generated using the system and the user then completes the application by providing the necessary sequential code.

Enterprise [24] is a complete programming environment for developing parallel applications. It supports the design, coding, debugging, testing, monitoring, profiling and execution of distributed parallel programs. Enterprise's use of patterns lies in its use of the analogy of a business organization or 'enterprise' to represent the parallelism of an application. A programmer describes the structure of parallelism using *assets*. The top-level abstraction is an *enterprise* which is composed of other assets such as a *line* to represent pipeline parallelism, a *department* to represent a work pool of processes, and a *division* to represent divide-and-conquer parallelism. The sequential processes (the pieces of work to do in parallel) are called *individuals* in the system and form the leaves of the *organizational chart* or description of the parallelism. One of the strengths of Enterprise is that it abstracts the communication and synchronization of the parallelism from the sequential code pieces. This allows the parallel structure of the application to be changed or tuned with minimal or no change to the existing sequential code.

While PAS and TACO are at a higher level of abstraction than Munin, they are not as high level as Enterprise. In PAS, the user explicitly specifies the communication and synchronization points in the computation, whereas in Enterprise these points are inserted into the code by the tool. Similarly with TACO, the user is required to specify the application topology and synchronization points, so its level of abstraction is also lower than Enterprise. Tracs and Enterprise are at a similar level of design abstraction, and both allow the user to specify the structure of a parallel application graphically.

2.2 Testing the Usability of Parallel Programming Systems

A set of thirteen characteristics for an ideal parallel programming systems was proposed by Singh *et al.* [26]. Some of the characteristics include:

Performance The system should provide the best performance possible given such factors as the combination of patterns, the generality of the patterns, and the level of abstraction of the patterns. This is often the primary criteria for assessment of parallel programming systems.

Usability How easy is the system to learn and use? In the analysis of such systems, this property is second only to performance in importance.

Utility A programming system should allow a range of different applications to be written using the system.

Extendible It should be possible to increase the number of patterns supported by a pattern-based system. This is a major failing with many pattern-based parallel programming systems in that they only supply a limited number of patterns and provide no way to add new ones.

Separation How well does a system separate the pattern code from the application code? This is the emphasis of most pattern-based programming systems.

Correctness The system should offer some guarantee as to the correctness of the generated code, such as deadlock avoidance.

The authours used these criteria to assess two programming systems, FrameWorks and Enterprise, which were both developed by their research efforts. Enterprise was shown to be an improvement over FrameWorks, but was still found to be lacking or weak in certain areas.

CO_2P_3S is the successor to Enterprise and has already been evaluated using these characteristics [18]. However, the portion of this dissertation which describes the utility of CO_2P_3S states that this aspect of CO_2P_3S is being improved and that new applications are being mined for new patterns or extensions to existing patterns. This improvement to the CO_2P_3S system is the work described in this dissertation.

A usability study of CO_2P_3S was also conducted as part of [18]. An undergraduate class was split into two groups with one group using the CO_2P_3S system and the other using Java threads directly for two assignments. After the first assignment, the tool that the students used was switched so that those who used CO_2P_3S became the group using threads and *vice versa*. Due to a number of problems with collecting data for the study, the results were generally inconclusive, although the study seemed to indicate that CO_2P_3S users wrote less code in general.

Another study examined the effect of pattern-based parallel programming tools in the context of the maintainability of applications written using these tools [28]. Thirty different software-engineering metrics were used to compare the implementation of a Mandelbrot Sets application. The results showed that there was a strong indication that the use of pattern-based parallel programming systems led to applications which were less complex and could be maintained with less effort.

2.3 The Cowichan Problems

Test suites for assessing the *performance* of a system, such as SPEC and SPLASH, abound in the computing world. In contrast, the number of test suites which address the *utility* or *usability* of a system are few. For parallel programming systems there is only one: the Cowichan Problems [31]. The Cowichan Problems is a suite of seven problems specifically designed to test the breadth and ease of use of a parallel programming tool as opposed to testing the performance of the programs that can be developed using the tool [32]. The goal of these problems is to provide a standard set of 'non-trivial' medium-size problems by which different parallel programming systems may be compared. These problems were part of a larger project whose goal was to assess parallel programming systems in two ways: how well the system can support large-scale software-engineering, and how easily a system can be learned.

The problem set is comprised of seven problems designed to test different aspects of a parallel programming system. The problems are from a wide selection of application domains and parallel programming idioms covering a range from numerical to symbolic applications, from data-parallelism to control-parallelism, from coarse-grained to fine-grained parallelism, and from local to global to irregular communication. The problems also address important issues in parallel applications such as load-balancing, distributed termination, non-determinism, and search overhead. Specifics of each of these problems will be given later in the context of the parallel pattern that can be used to solve the problem. For this work, one modification was made to the original problem set. The Active Chart Parsing problem was replaced by the Fifteen Puzzle. Details on why this substitution was made is deferred until Section 7.6. Therefore the applications used for this research are:

- Turing Ring Problem (Chapter 4)
- Image Thinning (Chapter 4)
- Computing Polygon Overlays (Chapter 5)
- Skyline Matrix Solver (Chapter 6)
- Matrix Product Chain Solver (Chapter 6)
- Game of Kece (Chapter 7)
- Fifteen Puzzle (Chapter 7)

The Cowichan Problems were used by a group at Vrije University (Amsterdam) to assess the Orca parallel programming language, which is "an imperative programming language based on a simple form of distributed memory"[32]. The problems were given to six Masters students who were in their final year, and each project was completed in three to seven months. The authors found that even though they had "ample experience with [Orca]", the problem set provided an adequate stress test for the language as the programmers needed to use many different aspects of the language. Flaws in the Orca language were exposed and the experience had a great impact on the further research of the language.

A second set of Cowichan Problems (the Cowichan II Problem set) is described in a later work of Wilson [33]. This second set of problems consists of fourteen problems, all of which "should require no more than an afternoon to write and test in a well-supported sequential language." Also, the programs can be chained together (that is, the output of one program forms the input of another program) in order to further test the code-reuse and modularization of the system. This second set of problems was developed in response to Wilson feeling that the time required for the first set (six to eight weeks per problem) was too "prohibitive" to allow for the wide acceptance of the original test suite. However, this is the exact reason that the first set of applications is used in this research, as those in the second set are "toy" problems which do not adequately simulate the real use of a system such as CO_2P_3S.

Given the insights into the Orca language that were achieved at Vrije University, it follows that similar insights into the CO_2P_3S system may be gained by implementing the problem set using the tool. Therefore, similar to the Orca group, this work uses the Cowichan Problems as a measure of the 'readiness' of the CO_2P_3S system for general use in the parallel programming community. Specifically, does CO_2P_3S have sufficient *utility* to implement solutions for the seven problems? If it does not, does CO_2P_3S have sufficient *extensibility* to add patterns that can be used to the solve these problems? Finally, does CO_2P_3S have sufficient *usability* that these solutions and any necessary CO_2P_3S extensions can be built in a reasonable time-frame?

Chapter 3

CO_2P_3S and MetaCO_2P_3S

3.1 The CO_2P_3S Parallel Programming System

The CO_2P_3S parallel programming system is a tool for implementing parallel programs in Java using the Parallel Design Patterns (PDP) methodology [18]. It addresses the two fundamental reasons that, to date, prevent parallel programming tools from being widely accepted: performance and utility. The first issue comes from tools which produce code that is too generalized to produce good parallel performance, and typically provides no support for further tuning the code. The second issue comes from tools which only produce efficient code for a small specific set of applications and provide no way to extend the system to meet the needs of other applications.

CO_2P_3S addresses the first issue through the use of *pattern templates*. A pattern template is an intermediary form between a pattern and a framework, and represents a parameterized family of design solutions. Members of the solution family are selected based upon the values of the parameters for the particular pattern template. For example, the shape of the matrix used for computation in the Wavefront pattern (Chapter 6) has three values (full, banded, triangular) and the particular value of the parameter significantly affects the framework code generated. This is where CO_2P_3S differs from other pattern-based parallel programming tools. Instead of generating an application framework which has been generalized to the point of being inefficient, CO_2P_3S produces a framework which accounts for application-specific details through the parameterization of its patterns. For the Wavefront pattern, CO_2P_3S generates one of these different (but related) frameworks based on the matrix shape.

A framework generated by CO_2P_3S provides the communication and synchronization for the parallel application, and the user simply provides the application-specific sequential code. These code portions are added through the use of sequential *hook methods* in the framework code. Recall that a hook method is a method which is overridden in the framework code to provide application-specific functionality. This abstraction of parallelism from the application-specific portions maintains the correctness of the parallel application since the user cannot change the code which implements the parallelism.

CO_2P_3S addresses the performance tuning issue by providing an open programming model based on descending layers of abstraction. The first layer, the *Patterns Layer*, is the highest level of abstraction and is where pattern templates are used. The next layer is the *Intermediate Code Layer*, which describes the parallelism of the program in a high-level, object-oriented, explicitly-parallel language. The final layer is the *Native Code Layer*, which is a translation of the structures in the previous layer into a native object-oriented language such as Java or C++. Programmers can progressively tune their application through these layers in order to meet performance needs.

The pattern parameters in CO_2P_3S can be divided into four types of parameters: lexical, design, performance, and verification parameters. *Lexical parameters* are various class and

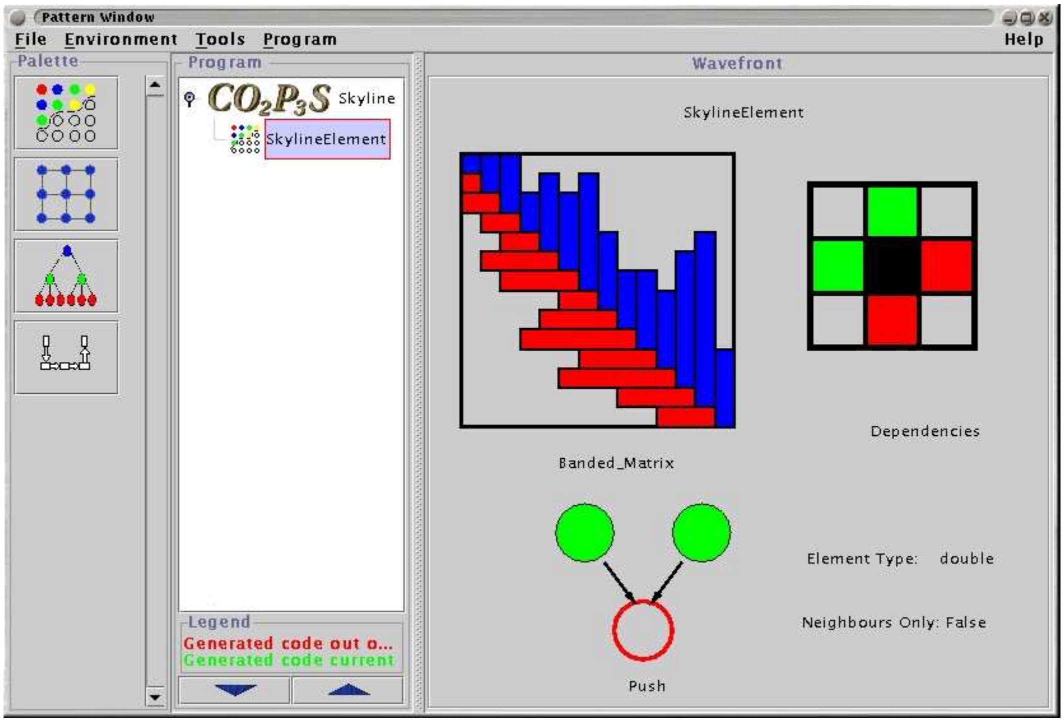


Figure 3.1: The CO_2P_3S graphical user interface. The Pattern Pane is on the right-hand side.

method names in the pattern framework which are provided by the user. For example, the name of the class representing a node in the Search-Tree pattern (Chapter 7) is a lexical parameter. *Design parameters* are pattern parameters which affect the overall parallel structure of the generated framework. The matrix shape in the Wavefront pattern (Chapter 6) is a design parameter. *Performance parameters* do not affect the overall structure of the framework, but introduce optimizations to improve performance. The notification parameter in the Wavefront pattern is a performance parameter since it can alter the technique used for inform elements that their data dependencies have been satisfied, but will have no effect on the structure of the generated framework. *Verification parameters* allow for the inclusion of pieces of code in the framework to ensure its proper use and to find faults in the code of the user. The verification parameter in the Search-Tree pattern assures that the `done()` hook method properly indicates when a node has completed its computation. This type of parameter was added as a result of the research described in this dissertation.

The GUI for the CO_2P_3S system is shown in Figure 3.1. On the left-hand side (the Pattern Palette) are the icons representing the various patterns supported in CO_2P_3S. A pattern is selected for an application by clicking on its respective icon. Going from the top to the bottom, the patterns represented are Wavefront, Mesh, Search-Tree, and Pipeline. The center of the GUI (the Program Pane) shows the name of the application (the Skyline Problem from Section 6.7) and any patterns selected. Here the Wavefront pattern has been selected. The name shown for the pattern (SkylineElement) is the name given by the user for the template class in the Wavefront pattern. The right-hand side (the Pattern Pane) shows the interface for a selected pattern; the Wavefront pattern. Using pop-up menus from this pane, the user sets the parameter values for the specific pattern. Shown in this figure are the settings for one parameterization of the Wavefront pattern.

Figure 3.2 shows the Program Options Pane for the same application. Here the user

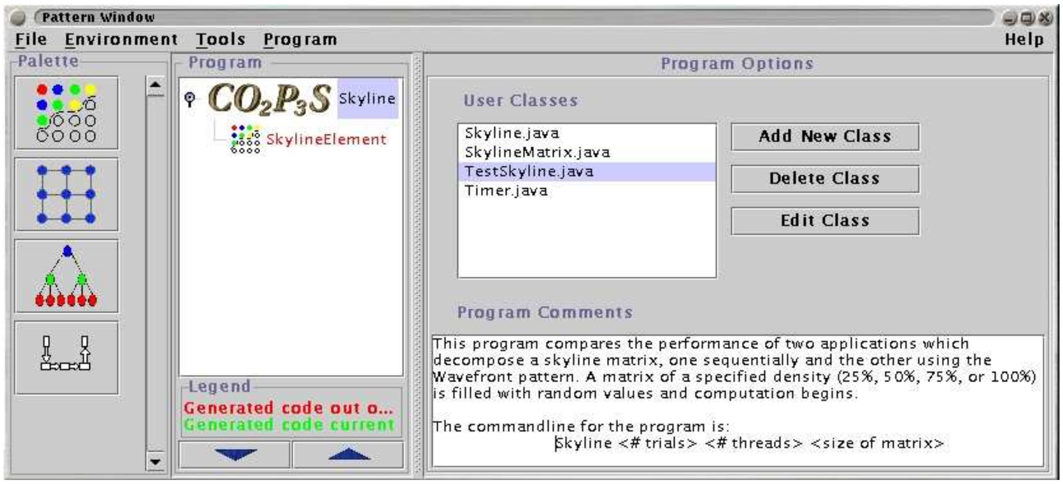


Figure 3.2: The CO_2P_3S graphical user interface. The Program Options Pane appears on the right-hand side.

includes any application-specific classes required for the application, as well as any comments for the application. As CO_2P_3S promotes the reuse of code, the classes which the user includes here are typically the same classes used in the sequential version. Comments by the programmer may include a description of the application, how to run the application, or information useful during maintenance of the program.

The user adds the code for the hook methods of a pattern using the Pattern Template Viewer as seen in Figure 3.3. Each pattern in CO_2P_3S contains one or more classes which require the user to insert code for hook methods. The pattern designer specifies which portions of the class are editable, and the viewer uses hyperlinks to restrict the user to only those portions. This prevents the user from accidentally altering any framework code in the template class, such as changing a method signature. Figure 3.3 shows a portion of the template class for the Wavefront pattern. There are three hyperlinks shown (the items underlined in Figure 3.3) which allow the user to import any necessary packages and/or classes, implement the *initialize()* method, or implement the *reduce()* method. When the user selects a hyperlink, a new window is opened that can be used for text entry. When the user closes that window, whatever modifications were made at that point are reflected in the template viewer if the "Regenerate After Each Change" box is selected.

The framework for a pattern in the application is generated via a pop-up menu in the Pattern Pane. Once the framework is generated and any necessary application classes have been added, the user can compile and run a program via the Program menu. The Program menu provides access to two dialogue windows: the Compile dialogue (Figure 3.4(a)) and the Run dialogue (Figure 3.4(b)). The Compile dialogue provides common compilation options via checkboxes, as well as letting the user add any specific compilation flags. The user can also remove all class files through the Clean button, or stop a compilation via the Abort button. The output from the compilation is displayed in the text window at the top.

After compilation, the Run dialogue allows the user to run an application. There are various options which the user may select, such as whether the program is to use shared or distributed memory, the heap and stack size of the virtual machine, and whether to use green or native threads. The user can also add any runtime flags via the Flags field. In order to run the program, the user fills in the bottom text field with the appropriate class name and command-line parameters.

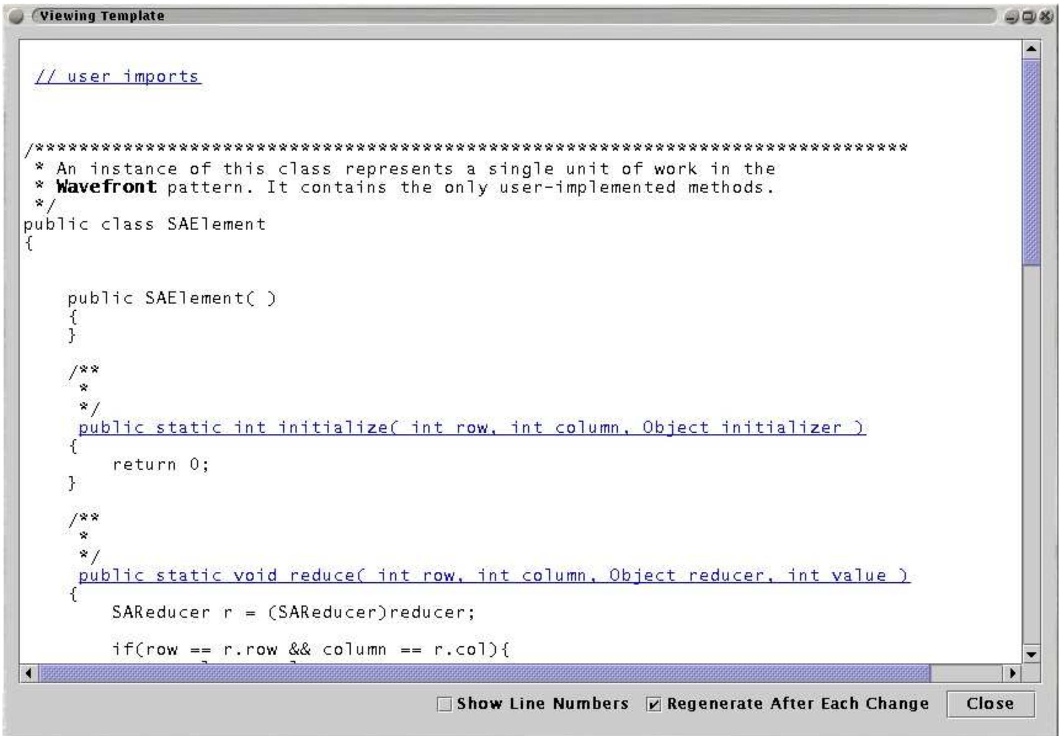
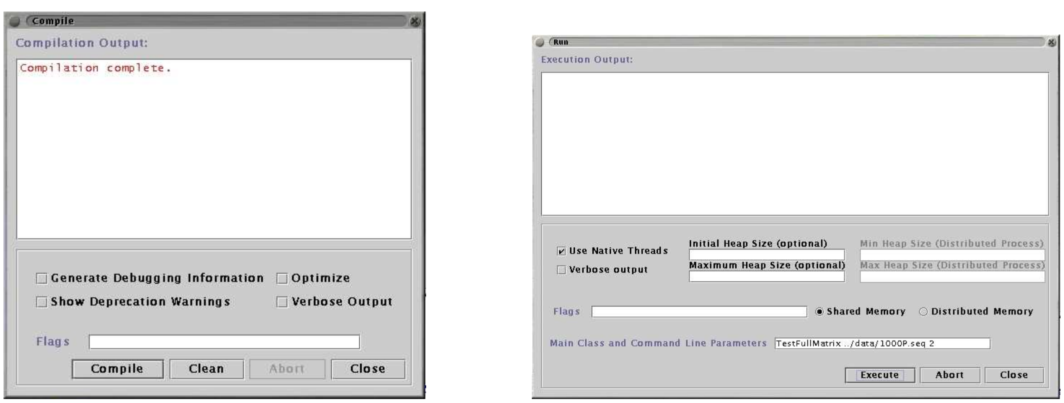


Figure 3.3: The template class viewer in CO_2P_3S.



(a) The Compile dialogue in CO_2P_3S.

(b) The Run dialogue in CO_2P_3S.

Figure 3.4: Compiling and running programs in CO_2P_3S.

3.2 The MetaCO_2P_3S Tool

CO_2P_3S allows new pattern templates to be added to the system by using the tool *MetaCO_2P_3S*. Pattern templates added through MetaCO_2P_3S are indistinguishable in form and function from those already contained in CO_2P_3S [7]. This allows CO_2P_3S to adapt to the needs of the user; if CO_2P_3S lacks the necessary pattern for a problem then the user is free to add a new pattern.

The MetaCO_2P_3S tool guides a *pattern designer*[1] through the creation of a pattern template. Pattern templates are constructed from three pieces of information:

1. *The names of the classes used in the framework.* These names are place holders that are replaced at run-time with unique names, since multiple instances of a pattern template may be used within the same application. The unique names are based on at least one user-supplied class name (a lexical parameter) for each instance of the pattern used in the application.

2. *The parameters for the pattern template.* MetaCO_2P_3S currently provides three parameter types:

   Basic Parameters These parameters represent an enumerated list of choices for a particular parameter, and are the most common type of parameters.

   Extended Parameters These parameters represent parameter values of an arbitrary form and are the least common type. For this type of parameter, the pattern designer must provide extra information such as how to collect the settings for the parameter and how each parameter affects code generation.

   Extended List Parameters These parameters represent a common instance of extended parameters in which the pattern user is to supply a list of values which may in themselves be basic or extended parameters. An example of this parameter type is a list of user-defined methods to be called in one of the framework methods.

3. *The GUI configuration for the pattern template representation in the CO_2P_3S GUI.* The GUI elements which can be specified in MetaCO_2P_3S are either text labels or images. These typically are dependent on the value of a parameter so that the pattern GUI accurately reflects the parameter settings.

As CO_2P_3S generates a customized framework based on the selection of a pattern template and a specific set of values for its parameters, the pattern designer must provide a framework template that defines the code to be generated. Correctness of the framework code is the responsibility of the pattern designer. Each framework source code template is annotated to provide the necessary information for proper generation of the framework. These annotations consist of:

- Placeholder names of the framework classes to be replaced at runtime with unique names as provided by the user through lexical parameters.

- Variables, methods, and portions of methods which are selectively generated based on the value of basic parameters.

- Additional methods which are generated based on extended (or extended list) parameter settings.

- Framework classes which contain the hook methods that allow application-specific code to be added by the user.

[1] A parallel and object-oriented programming expert who creates new pattern templates.

```
/**
    Description of the following Java method

    @sampleTag a tag that is parsed by Javadoc
*/

public void methodName()
```

Figure 3.5: A Javadoc comment.

Generation of the framework is accomplished using Javadoc, a tool included with Java distributions for generating API documentation. Javadoc runs a modified compiler on Java source code files to parse declarations and specially formated comments to generate the documentation for the given classes. The form of a Javadoc comment is shown in Figure 3.5.

Javadoc has been extended to allow pluggable *doclets* to be used. A doclet is a Java program which satisfies a contract so that it is allowed to receive the parsed data from a Javadoc execution. This includes the declarations from each parsed class, but excludes method bodies and field initializations which are ignored by the Javadoc parser. MetaCO_2P_3S defines a set of Javadoc tags and macros which are used for the annotation of the framework. These are used in conjunction with a doclet to produce the generated framework. As a given Javadoc parse does not contain the method bodies, the bodies of methods are stored in separate files from the declarations. When a CO_2P_3S framework is generated, the annotated declarations and appropriate method bodies are merged based on the parameter settings.

The descriptions of pattern templates generated by MetaCO_2P_3S are stored in system-independent XML[2] format. This allows for patterns generated by MetaCO_2P_3S to be used not only by the CO_2P_3S system itself, but also by any template-based programming tool which uses XML. When a pattern template is imported into CO_2P_3S, the XML description is converted into a compiled plug-in module, and XSL[3] is used to transform the pattern description into a Java source file. This file is then compiled and loaded into CO_2P_3S.

Part of the MetaCO_2P_3S GUI-based description of the Wavefront pattern is shown in Figure 3.6. The *Constants* section contains the definition of the constants used in the pattern. These are typically the values of the basic parameters and text which are displayed in the Pattern Pane. The nest section shows the names of the various classes in the pattern. Framework classes, those to which the user has no direct access, are shown with square braces. All the names for the framework classes here are based on the name of the one template class (WavefrontElement) which the user specifies through a lexical parameter. The third section shows the parameters for the Wavefront pattern. Most of these are basic parameters, with the parameter *dependencies* being an extended parameter. Finally, the GUI configuration specifies the look of the Pattern Pane and how it changes based on the parameter settings.

[2] Extensible Markup Language.
[3] Extensible Stylesheet Language.

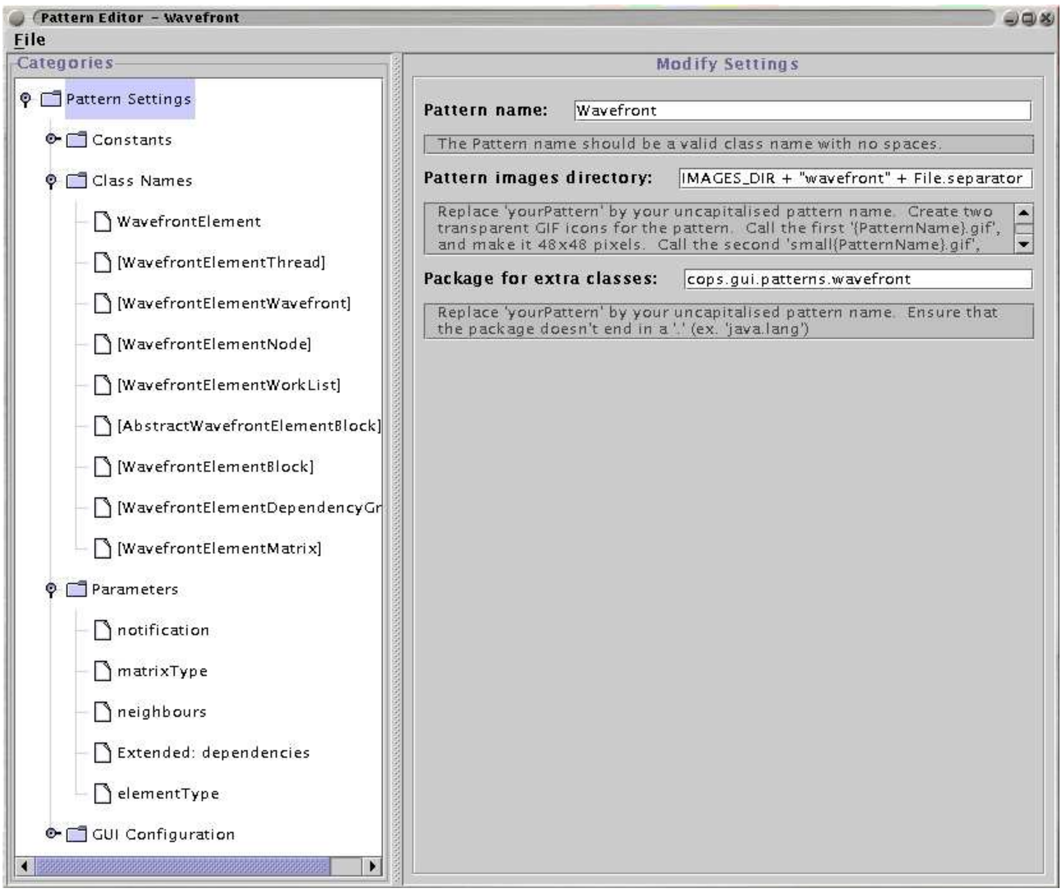


Figure 3.6: The Wavefront pattern in MetaCO_2P_3S.

Chapter 4

The Mesh Pattern

4.1 Overview of the Mesh Pattern

The Mesh pattern is used for computing elements of a regular, rectangular two-dimensional data set where each element is dependent on its surrounding values. In other words, it is used for applications where the elements are evenly spread over a two-dimensional surface and computation of an element is dependent on values from either the cardinal points or from all eight directions. Figure 4.1(a) shows such a mesh. This class of application includes programs for weather prediction and particle simulation.

In an application which uses a Mesh pattern, the elements are repeatedly iterated over to produce new values, which are based on the value of the element and the surrounding elements, until a certain condition is met. Programs which use the Mesh pattern may use either Gauss-Seidel or Jacobi iterations, but each element must be able to determine strictly from its local state when it is has reached its final value. Aside from data dependencies between neighbouring elements, no other dependencies between elements can exist [18].

The parallelization of an application which uses a mesh is accomplished by spatially decomposing the mesh into partitions and performing one iteration in parallel on all the partitions, as shown in Figure 4.1(b). Boundary values are then exchanged between partitions and another iteration is done. This continues until the local stopping condition is satisfied for all elements [18].

4.2 Chapter Overview

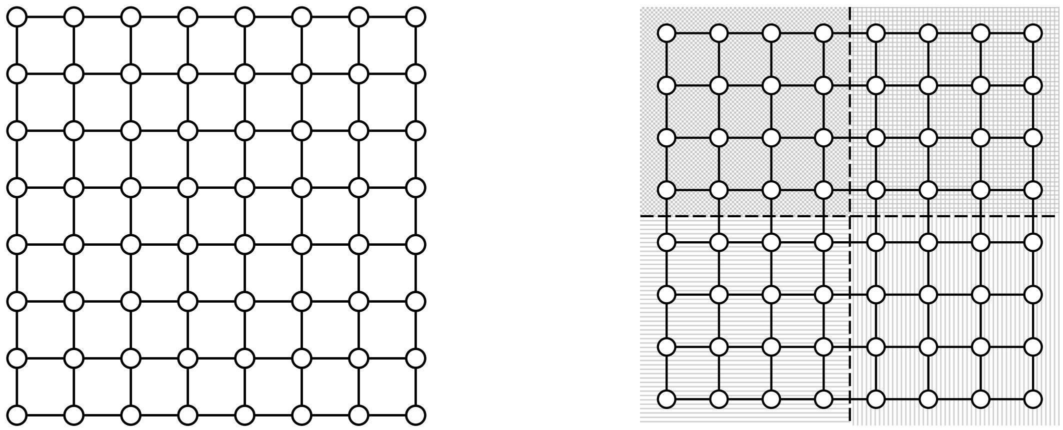
This chapter begins with a description of the Mesh pattern in CO_2P_3S. Next, the Image Thinning and Turing Ring problems, two problems from the Cowichan Problem Set that can be implemented using a Mesh pattern, are discussed. For each of the problems, a description of how the pattern was used to solve the problem and results of using the pattern are presented. Finally, a summary of the chapter is given.

4.3 The Mesh Pattern in CO_2P_3S

The Mesh pattern was one of the first patterns supported by CO_2P_3S. A view of the pattern in the CO_2P_3S system is shown in Figure 4.2. The figure shows the default configuration of the pattern before it is parameterized for a specific application.

4.3.1 Pattern Parameters

The pattern contains three lexical parameters: the Mesh class name, the MeshElement name, and the MeshElementSuperclass name. The Mesh class represents the entire mesh



(a) A mesh.

(b) Decomposition of a mesh.

Figure 4.1: A regular, rectangular two-dimensional mesh.

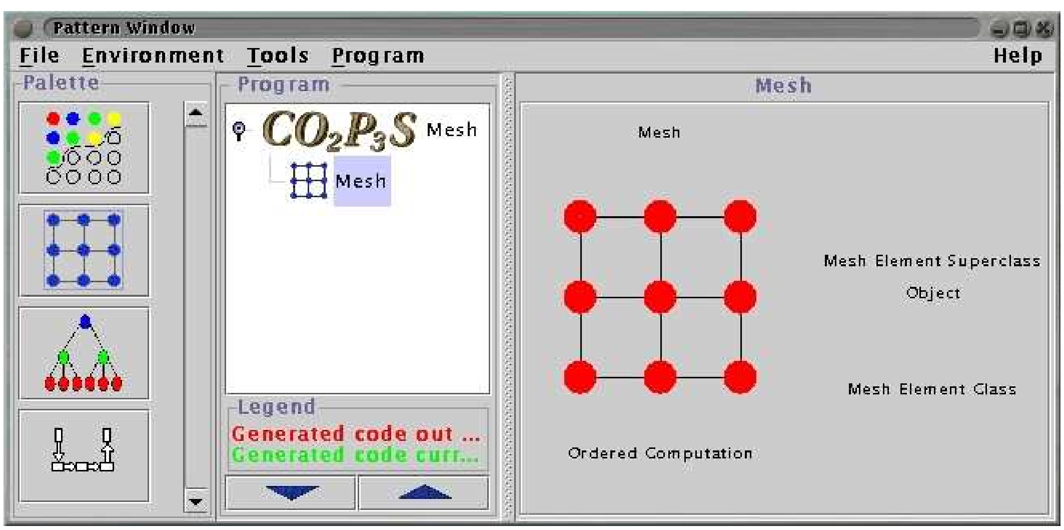


Figure 4.2: A screenshot of the Mesh pattern in CO_2P_3S.

in the application and is used to initiate the computation. The MeshElement represents a single element of computation in the mesh, and the MeshElementSuperclass is its superclass (typically the class Object).

There are two design parameters in the Mesh pattern. The first is the *boundary conditions* parameter which dictates how the edge elements of the mesh are handled. The values for this parameter are:

Non-toroidal, where none of the edges wrap around. This is the default value.

Fully-toroidal, where all of the edges wrap around to the opposing edge. This means that all elements of the mesh have access to all of their neighbours.

Horizontal-toroidal, where only the elements on the horizontal edges wrap around. This means that elements on the left and right edges (excluding corners) have access to all of their neighbours, but elements on the top and bottom (including corners) do not receive data from their bottom and top neighbours.

Vertical-toroidal, which is the horizontal toroidal rotated 90°.

The second design parameter is the *number of neighbours* parameter. This specifies whether an element requires data from its neighbours only at cardinal points (a 4-point mesh) or from all eight of its neighbours (an 8-point mesh). The 4-point mesh is the default setting.

The performance parameter in the Mesh pattern is the mesh ordering which indicates the amount of synchronization generated in the framework. The ordering can be either *ordered*, which performs Jacobi iterations or *chaotic*, which performs Gauss-Siedel iterations. If Jacobi iterations are used then the current iteration uses only the values computed in the previous iteration. In contrast, Gauss-Siedel uses the values computed in current iteration as well as those from the previous iteration. The default value creates an ordered mesh.

4.3.2 Hook Methods

Depending on the values for the design parameters (*boundary conditions* and *number of neighbours*), up to nine operation hook methods may be generated in the Mesh element class. For a particular element, the correct operation method is called based upon its location in the mesh. The methods generated for a non-toroidal 8-point mesh are shown in Figure 4.3, as this setting produces the widest variety of hook methods. The 4-point mesh produces a subset of these methods with appropriate parameters. Recall that `MeshElement` is a lexical parameter and will be replaced by the user-specified class name.

In addition to the operation hook methods in Figure 4.3, the hook methods of Figure 4.4 are also generated. These methods are generated regardless of the mesh configuration specified by the parameter values. All hook methods appear in the MeshElement class.

The hook method constructor is used by the framework to create the elements of the mesh. It is called by the framework when the Mesh object is created. As shown in Figure 4.4, the MeshElement constructor takes as parameters the location of the element (`i` and `j`), the width and height of the mesh (`dataWidth` and `dataHeight`), and a user-provided `initializer` object used for any initialization of the element. Similarly, the `reduce()` method is used at the end of the mesh computation to gather results from the mesh. The parameters for this method are the same as for the constructor, except that a user-provided reducer object is passed to the method instead of an initializer object.

The other four hook methods (`initialize()`, `notDone()`, `prepare()`, and `postprocess()`) are used during the processing of the mesh. The `initialize()` method is used to do any initialization of the mesh elements that may be done in parallel. Its complement, `postprocess()`, is used to do any post-processing of elements which can be done in parallel. The `notDone()` method signals the completion status of a mesh element. If `false` is

```
topLeftCorner(MeshElement east, MeshElement southeast,
        MeshElement south)
topEdge(MeshElement east, MeshElement southeast,
        MeshElement south, MeshElement southwest,
        MeshElement west)
topRightCorner(MeshElement south, MeshElement southwest,
        MeshElement west)
leftEdge(MeshElement north, MeshElement northeast,
        MeshElement east, MeshElement southeast,
        MeshElement south)
interiorNode(MeshElement north, MeshElement northeast,
        MeshElement east, MeshElement southeast,
        MeshElement south, MeshElement southwest,
        MeshElement west, MeshElement northwest)
rightEdge(MeshElement north, MeshElement south,
        MeshElement southwest, MeshElement west,
        MeshElement northwest)
bottomLeftCorner(MeshElement north, MeshElement northeast,
        MeshElement east)
bottomEdge(MeshElement north, MeshElement northeast,
        MeshElement east, MeshElement west,
        MeshElement northwest)
bottomRightCorner(MeshElement north, MeshElement west,
        MeshElement northwest)
```

Figure 4.3: The operation methods for the Mesh pattern.

returned by this method then processing of the the element is complete, otherwise computation of the element requires further iterations. Threads iterate over their elements and exchange the completion status of their partition with other threads in order to determine when computation is globally completed. The `prepare()` method is called prior to the operate methods to allow for any necessary preprocessing of elements.

In computing the mesh, each thread performs the loop shown in Figure 4.5 which ensures proper computation and communication of values between blocks of mesh elements. The `notDone()`, `prepare()`, and `operate()` iterate over all elements in the block and calls the hook method of the same name. The `synchronize()` method provides the necessary synchronization between partitions as dictated by the setting of the ordering parameter. If the value of the ordering parameter is *ordered* then the `synchronize()` method acts as a barrier. If the value is *chaotic* then the method does nothing. The `operate()` method calls

```
MeshElement(int i, int j, int dataWidth, int dataHeight,
    Object initializer)
reduce(int i, int j, int dataWidth, int dataHeight,
    Object reducer)
initialize()
notDone()
prepare()
postprocess()
```

Figure 4.4: Hook methods for the Mesh pattern.

```
this.initialize();
while(this.notDone()){
    this.prepare();
    this.synchronize();
    this.operate();
}
this.postProcess();
```

Figure 4.5: The main loop for each thread in the Mesh pattern.

the appropriate hook method from Figure 4.3 for each element of the partition.

4.3.3 Using the Pattern

The Mesh pattern is used in an application by instantiating a Mesh object and sending that object the `launch()` message. The constructor of the Mesh object takes as arguments the following: the width and height of the mesh, the number of partitions vertically and horizontally, an initializer object (or `null` if not required), and a reducer object (or `null` if not required). A thread is created for each partition of the mesh. The `launch()` method will return when the mesh computation is finished and the final results are available via the reducer object (if one was provided).

4.4 Image Thinning

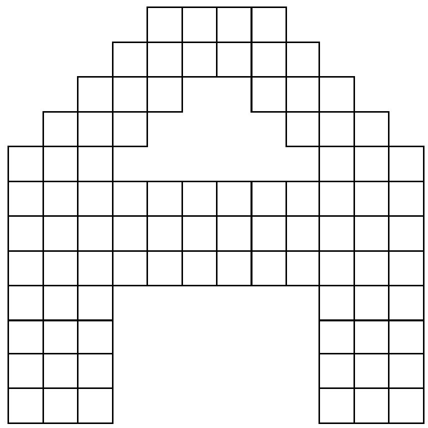
4.4.1 Description of the Problem

Image thinning is an important stage in the processing of images such as electron density maps of proteins and handwriting or character recognition. The Image Thinning problem takes an image which contains straight-line segments of varying widths, and thins the image so that lines have unit width [10, 31]. Figures 4.6(a) and 4.6(b) show an image before and after this process. While understanding the image thinning process is intuitively easy, providing a precise mathematical description is complicated. As a result, many thinning algorithms are described in the image processing literature, and each produces a different result for the same image. Figures 4.6(b) and 4.6(c) show two possible thinnings of the image in Figure 4.6(a).

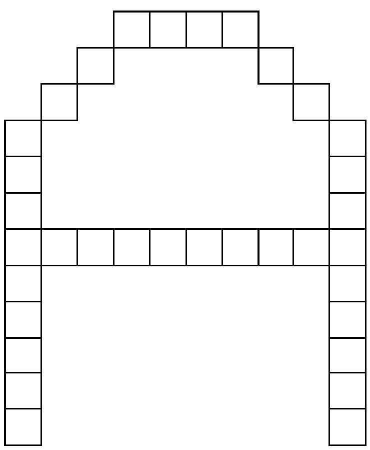
The input to the image thinning process is a two-dimensional image. Thinning of an image is accomplished by repeatedly passing over the image and removing pixels from the image unless they satisfy one of the following criteria:

1. The pixel is at the tip of a line segment.
2. Deleting it would disconnect an image component.

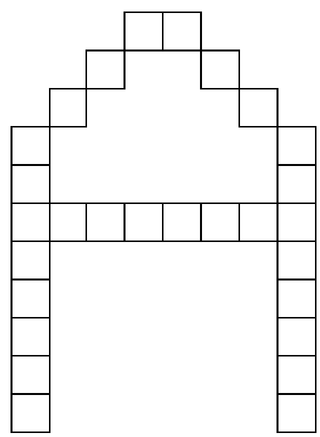
The removal of pixels is accomplished by applying a set of masks to a pixel and its surrounding neighbours. If the mask fits, then the pixel may be removed. Figure 4.7 shows a sample of masks used in the process of thinning an image. A one represents a pixel which is either darker than or the same colour (i.e. greater than or equal to) as the pixel under consideration. Similarly, the zeros represent pixels that are lighter or have a lesser value than the pixel under consideration. For example, in the "north" mask case the center pixel can be removed if all of the pixels above it are greater than or equal to the pixel itself and the pixels next it.



(a) An image before thinning.

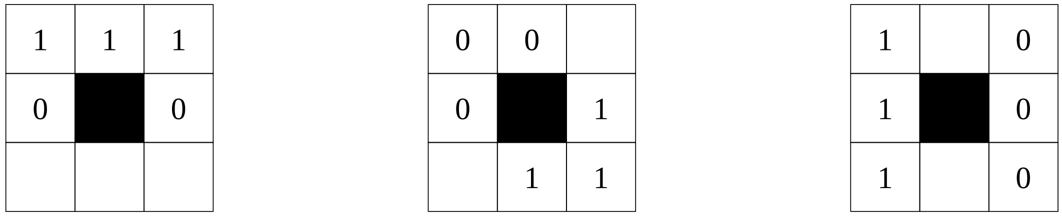


(b) The image after thinning.



(c) The image after thinning with a different algorithm.

Figure 4.6: An example of image thinning.



(a) A "north" pixel mask.

(b) A "south-east" pixel mask.

(c) A "west" pixel mask.

Figure 4.7: Pixel masks used in image thinning. The ones indicate pixels which are greater than or equal to the value of the pixel under consideration (the center square) and zeros indicate pixels that are less than this pixel.

A situation may occur in which either of two pixels may be removed. While this situation is easy to handle in the sequential case (sweep order is used), parallel implementations must take care to ensure that both pixels are not deleted. A characteristic of the use of pixel masks is that this situation will not occur.

4.4.2 Why the Mesh Pattern is Appropriate

Parallel implementations of image thinning typically divide the image into tiles with communication occurring between neighbouring tiles. As the thinning process performs multiple passes over the same image with information being passed along the boundaries of image tiles, the problem fits nicely into a Mesh pattern.

4.4.3 Using the Mesh Pattern

The parameterization of the Mesh pattern for this application is shown in Figure 4.8. The Mesh class is ThinImage, the MeshElement class is Pixel, and the MeshElementSuperclass is Object (the default value). An 8-point non-toroidal ordered mesh is selected for the structure of the mesh.

The hook method constructor copies a pixel value from an array of stored image pixel values (the initializer object). Since the Pixel object is initialized on construction, the `initialize()` method is left empty. The `reduce()` method performs the inverse of the constructor and copies the pixel values back into the image array (the reducer object). If the value of a pixel has changed in the last iteration, then `notDone()` returns `true`. A Pixel object contains two values: a read value and a write value. During computation of the write value, the read value of surrounding pixels is used. This implements an ordered computation (Jacobi iteration). The `prepare()` method copies the write value to the read value in preparation for the next iteration. The `operate()` hook methods are implemented using existing code taken from the sequential program. No post-processing of the pixels is required, so the `postProcess()` method is left empty.

4.4.4 Results

The sequential image thinning application written consisted of a single class 221 lines in length, of which 170 were reused in the parallel application to implement the hook methods and driver program. The parallel application consisted of 7 framework classes and 1 user class. The user class was a modified version of the sequential program, which was changed

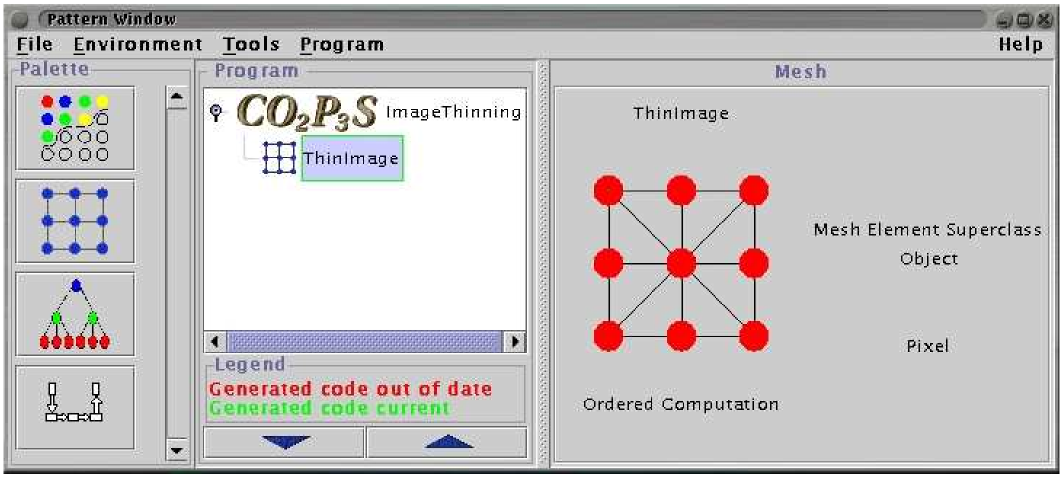


Figure 4.8: Parameterization of the Mesh pattern for an image thinning application.

to instantiate and launch the mesh computation. The parallel application contained 529 lines, of which 350 were from the generated framework and 9 were new.

The results of thinning a 3000 × 3000 pixel image are shown in Table 4.1. The times given are the median of ten runs. The application was run on an SGI Origin 2000 with 46 MIPS R100 195 MHz processors and 11.75 gigabytes of memory. A native threaded Java implementation from SGI (Java 1.3.1) with optimizations and JIT turned on was used and the virtual machine was started with 1 GB of heap space. The application performance obtains fair speedups for this data set. However, there appears to be a tapering off of performance toward 16 and 32 processors. This is a result of a decrease in the granularity of work as the number of processors increases.

Processors	1	2	4	8	16	32
Time (sec)	709	377	201	111	68	50
Speedup	-	1.88	3.53	6.39	10.43	14.18

Table 4.1: Speedups and wall clock times for thinning a 3000 × 3000 pixel image. The times given are the median of ten runs.

4.5 The Turing Ring Problem

The Turing Ring problem has previously been described and implemented in CO_2P_3S as a reaction-diffusion problem. It has appeared in several CO_2P_3S publications as an example of the Mesh pattern [18, 19, 20]. Therefore, only an overview of the problem and why the Mesh pattern is appropriate is given here. A summary of the results of using the Mesh pattern to solve this problem is also given. Actual performance results may be found in the previously mentioned publications.

4.5.1 Description of the Problem

The original problem formulated by Alan Turing modeled the interaction between two chemicals in a ring of cells through the use of two differential equations [30]. The problem was

$$dX_i/dt = X_i(\rho_X + \alpha_X X_i + \beta_X Y_i) + \mu_X(X_{i+1} + X_{i-1} - 2X_i) \tag{4.1}$$

$$dY_i/dt = Y_i(\rho_Y + \alpha_Y X_i + \beta_Y Y_i) + \mu_Y(Y_{i+1} + Y_{i-1} - 2Y_i) \tag{4.2}$$

Figure 4.9: Differential equations used in the Turing Ring model.

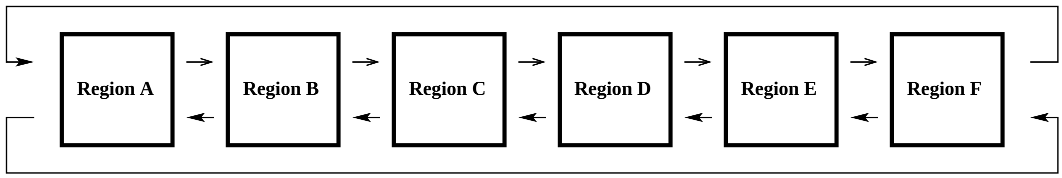


Figure 4.10: Example of a Turing Ring system.

later generalized to include other reaction-migration type problems, such as those found in ecology, epidemiology, and petroleum engineering [21]. Figure 4.9 shows the pair of coupled differential equations which model the populations X_i and Y_i for each cell in the ring.

For a predator/prey system, $\rho_{X,Y}$ represents the birth rate of the two species, $\alpha_{X,Y}$ is a constant death rate, $\beta_{X,Y}$ signifies deaths due to overpopulation of predators or the consumption of prey, and $\mu_{X,Y}$ is the migration rate between neighbouring cells [21]. By varying these same coefficients, the model can be used to represent the mixing of water and oil in porous rock, or the progression a disease through a population.

4.5.2 Why the Mesh Pattern is Appropriate

As shown in Figure 4.10, a Turing Ring can be represented as a set of adjoining cells where the last and first cells are adjacent to create the ring. As each cell requires data from the two adjoining cells and all cells are computed in a lock-step manner, the problem fits well with a Mesh pattern.

4.5.3 Summary of Results from Previous Publications

The Mesh pattern was used to implement a reaction-diffusion application. The program was run using a 1680 × 1680 surface on an SGI Origin 2000 with 44 195 MHz R10000 processors and 10 GB of memory. The JVM was started with 512 MB of stack space and optimizations turned on. The application was found to scale well up to 4 processors, but performance declined after that due to decreasing granularity. This same effect was seen with the Image Thinning application.

4.6 Summary

The Mesh pattern was used to implement solutions for two problems from the Cowichan Problems: the Turing Ring Problem and Image Thinning. The Turing Ring problem was previously described and implemented in other publications as an example of the use of patterns in CO_2P_3S, so only a description of the problem, why the Mesh pattern is appropriate, and a summary of published results was given. The reaction-diffusion application which was implemented was found to produce good speedups up to 4 processors, but performance declined after that due to granularity issues. The image thinning application scaled well up to 16 processors for a 3000 × 3000 pixel image, but as with the Turing Ring problem the granularity became an issue after 16 processors.

Chapter 5

The Pipeline Pattern

5.1 Overview of the Pipeline Pattern

Pipelines are common in both real-life situations, such as an assembly lines in a factory, and computing applications, such as the instruction pipeline in most modern processors. Pipelines provide a simple way of improving the performance of a task by separating a task into stages, each of which can be done in parallel. For example, instruction pipelines in modern processors improve program performance and resource usage by allowing both integer and floating-point instructions to be processed concurrently.

Abstractly, a pipeline can be viewed as a sequence of stages wherein the stages have a specific ordering between them so that the results of one stage forms the input for one or more following stages. Each stage of the pipeline can be viewed as having an object in a certain state, and transition between pipeline stages is simply a change of state for the object [18]. Figure 5.1 shows an example of a compilation pipeline. In this pipeline the source code file changes state from source code to abstract syntax tree and intermediate representation in the compiler to assembly language to machine code.

Traditionally pipelines are parallelized by assigning one or more threads to each stage of the pipeline. However, this can lead to load imbalances as some stages may require more computation and these particular stages may vary during a run of the application. The Pipeline pattern resolves this problem by taking a work-pile approach to the computation of pipeline stages. Each stage of the pipeline can be viewed as having a buffer of items to be processed in that stage. Since the processing of an item in the pipeline may be viewed as a transformation from one state to another, in a work-pile approach threads search the buffers for work, transform items to their next state, and place them into the next buffer if further processing is required. In this way the load can be balanced across the pipeline.

5.2 Chapter Overview

This chapter continues with a description of the Pipeline pattern in CO_2P_3S and how the pattern in used in an application. This is followed by a description of the Map Overlay problem, how the Pipeline pattern is used to solve it, and the results of using the pattern.

5.3 The Pipeline Pattern in CO_2P_3S

The pipeline pattern is one of the four original pattern contained in CO_2P_3S [18]. Figure 5.2 shows the initial view of the pattern in CO_2P_3S.

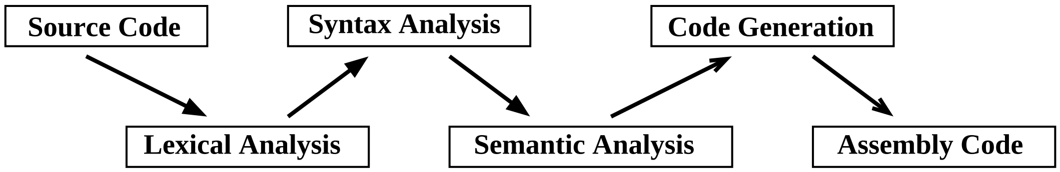


Figure 5.1: A compilation pipeline.

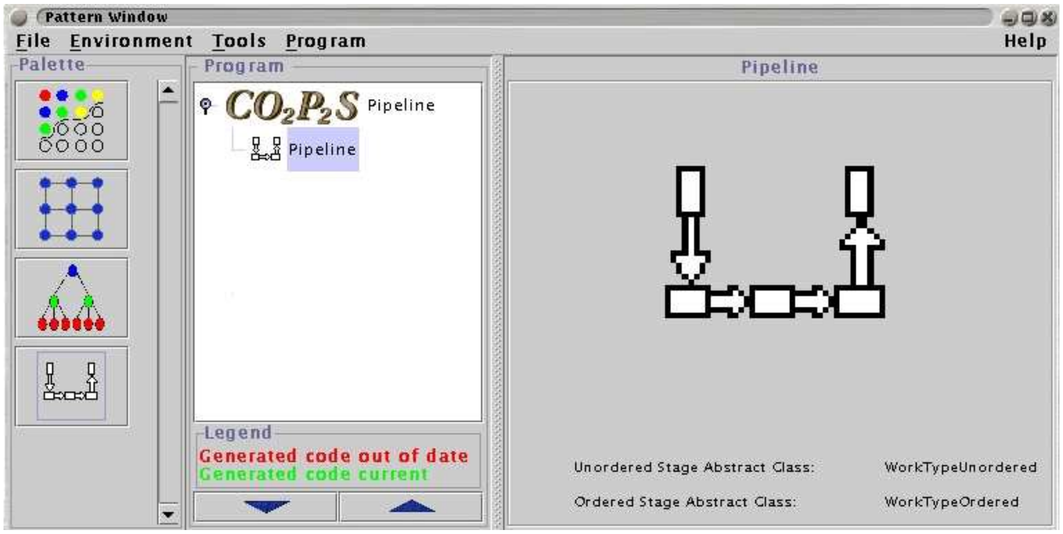


Figure 5.2: The Pipeline pattern in CO_2P_3S.

```
while(not finished)
    Find work from one of the buffers
    nextStage = work.execute()
    while(no buffer for output of nextStage)
        nextStage = nextStage.execute()
    Place nextStage into the appropriate buffer
```

Figure 5.3: Pseudocode for the computation loop in the Pipeline pattern.

5.3.1 Pattern Parameters

The Pipeline pattern is unlike the other CO_2P_3S patterns presented in this work in that the pattern is a structural pattern. In other words, the pattern does not contain any design, performance, or verification parameters, but only lexical parameters. Instead of producing a framework which is tuned for a particular application, the pattern generates the structural framework of a pipeline for the user and the user uses the pattern by subclassing specific framework classes in order to specify the stages of the pipeline.

The three lexical parameters for the pattern are:

1. The name of the class representing the pipe,
2. The name of the class representing an unordered work type, and
3. The name of the class representing an ordered work type.

The work types, or stages, in a pipeline may either be ordered or unordered. An ordered stage enforces an order for the processing of items so that the items at a particular stage will be processed in the order that the items entered the pipeline. An unordered stage removes this restriction and allows for items to be processed out of order in stages of the pipeline. The use of unordered buffers is an optimization of the pipeline and does not affect correctness of the pipeline computation. Therefore, a user is allowed to place an ordering on the work items only when it is necessary for correctness.

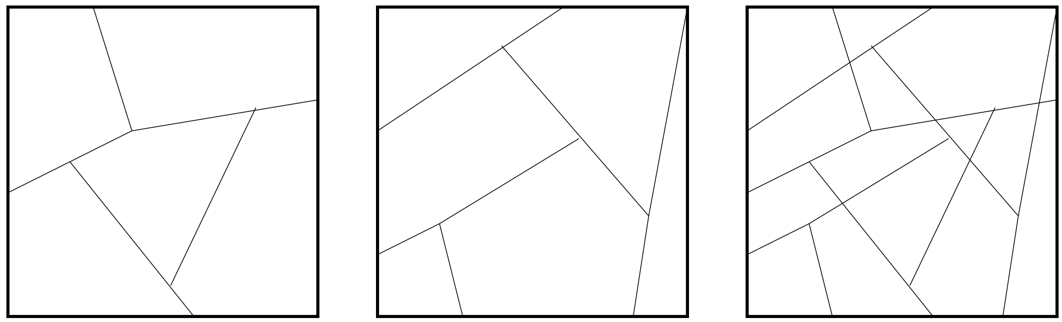
5.3.2 Hook Methods

The user is required to implement a single hook method for each stage of the pipeline. This is the `execute()` method which performs the computation of a particular stage.

Figure 5.3 shows pseudocode for the processing loop which is run by a thread in the Pipeline pattern. As long as there is more work to do, the thread will search the pipeline stage buffers for work items. If an item is found, then the `execute()` method is called. As long as the next stage of the pipeline does not require a buffer to enforce ordering constraints, then the thread continues processing the item. Otherwise the item is placed into the appropriate buffer.

5.3.3 Using the Pattern

To use the Pipeline pattern, the user creates the various stages of the pipeline by subclassing either the ordered or the unordered abstract pipeline stage for each stage. An application uses a Pipeline pattern by instantiating a pipeline object. The pipeline constructor has the form `Pipeline(int threads, Class[] workTypes)`. It takes the number of the threads to use in the application and an array of the classes of the pipeline stages which is used in connecting the stages of the pipeline. The pipeline object constructs the pipeline and begins the threads. Computation of items begins as soon as they are inserted into the pipeline via



(a) Map of tree type.

(b) Map of soil type.

(c) Map of tree and soil type.

Figure 5.4: An example of the map overlay problem.

the `addInputItem(WorkType)` method. Once all of the items have been placed into the pipeline, the resulting objects are collected using the `getResultItem()` method.

5.4 Generating Map Overlays

5.4.1 Description of the Problem

Given two maps A and B which both cover the same geographical area and are both decomposed into a set of non-overlapping polygons, what are the polygons produced by overlaying these maps? This problem regularly occurs in spatial information systems [11, 16, 31]. Imagine that map A represents types of trees and map B represents soil types. By combining these two maps, a new map is created which indicates the type of soil in which particular trees grow. Figure 5.4 gives an example of the overlay of two maps.

As the general case has many complications which do not contribute further to the problem's understanding, the following simplifications are made:

- All polygons are rectangles such as in Figure 5.5. This allows for easy storage of the polygons as only the top-left and bottom right-corners need to be stored.
- All vertices of the polygons lie on an integer grid to simplify arithmetic and generation of the maps.
- The two maps have the same extents or proportions.
- The maps are completely covered by their polygon decomposition such that there are no holes.
- Both maps can be held in memory to eliminate the effects of going to disk.

5.4.2 Why the Pipeline Pattern

The Pipeline pattern is used since the application may be viewed as the filtering of one map using the other map as a mask. This process is shown in Figure 5.5.

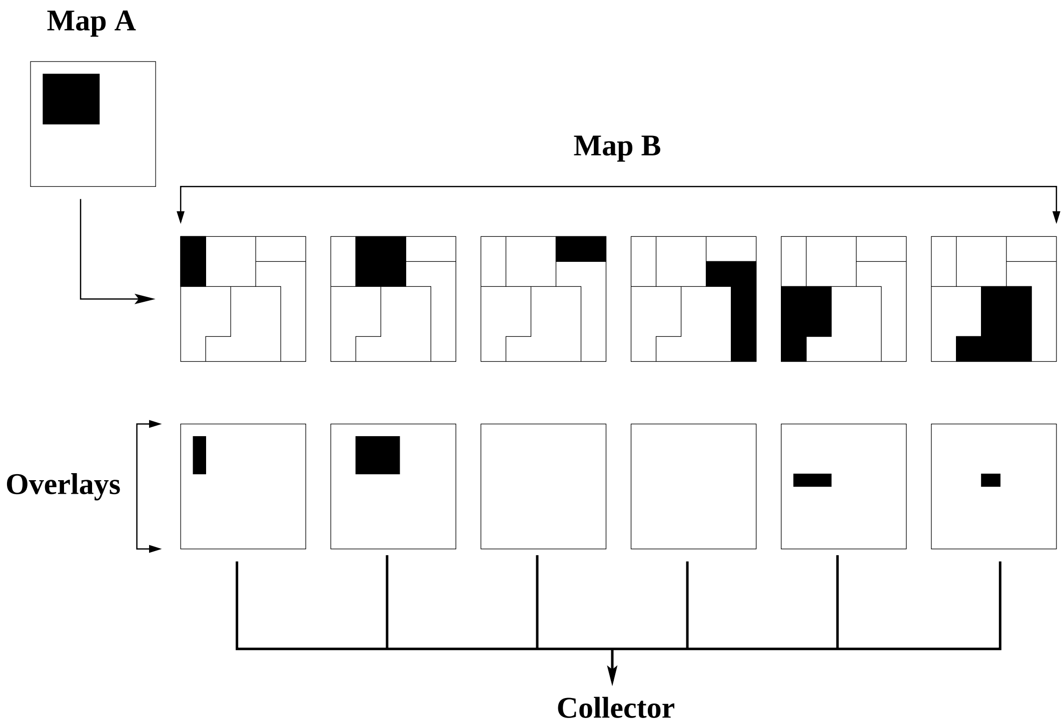


Figure 5.5: The filtering of polygon maps. Sections of Map A are compared with the sections of Map B and overlapping regions are collected.

5.4.3 Using the Pipeline Pattern

The specification of the pipeline in CO_2P_3S is shown in Figure 5.6. The pipeline for the Map Overlay problem consists of four stages, where each stage represents a quarter of Map A. Groups of regions from Map B are passed through the pipeline and at each stage are compared to the regions in Map A to find the overlapping regions.

As there is no ordering dependency within the stages (i.e. the work done in a stage is independent from the other work done in the same stage), the stages are unordered (i.e. the stages are subclasses of the unordered abstract class). Note that the ordered stage abstract class was specified in Figure 5.6. This was done so that the effect of using ordered versus unordered stages in this application could be assessed. As the pipeline and running time for this application are short, no difference in performance was observed between the use of the two types of buffers. In general, there can be a difference using a conventional pipeline, however the use of a work-pile paradigm helps to reduce this.

5.4.4 Results

The sequential version of the Map Overlay program consisted of 2 classes totaling 84 lines. In contrast, the parallel version which used the Pipeline pattern contained 38 classes and 423 lines of which 28 classes were from the framework and contained 228 of the lines of code. One class was reused from the sequential program and the 29 lines from the sequential program were reused. As the user had to create the classes for the various pipeline stages, the user had to write 135 new lines of code.

The application was run on an SGI Origin 2000 with 46 MIPS R100 195 MHz processors and 11.75 gigabytes of memory using a native threaded Java implementation from SGI (Java 1.3.1). Optimizations and JIT were turned on and the virtual machine was started with

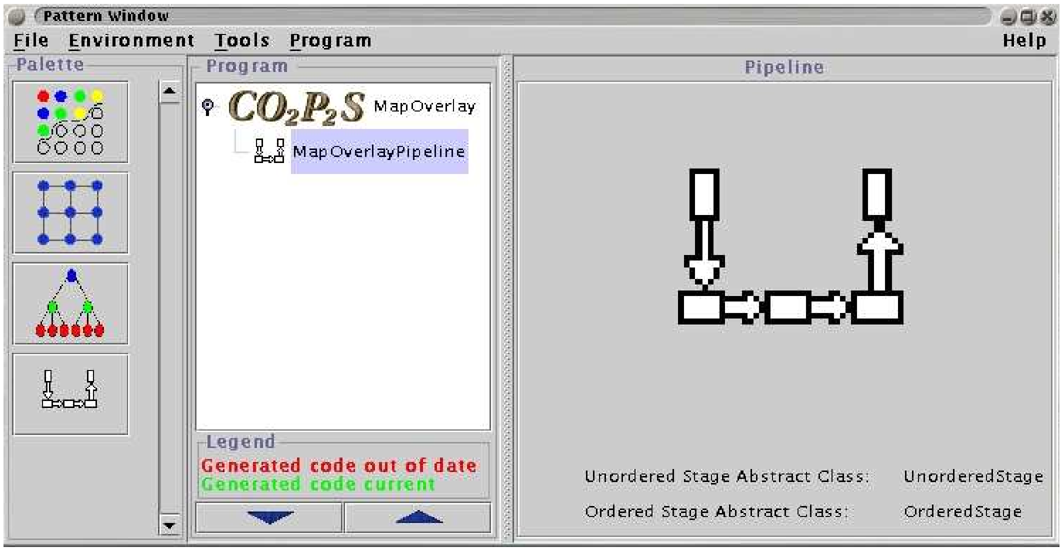


Figure 5.6: Configuration of the Pipeline pattern for a map overlay application.

1 GB of heap space. Table 5.1 shows the results of overlaying two images, one with 3000 horizontal stripes of unit height and one with 3000 vertical strips of unit width. These maps were used as they produce the greatest possible amount of work. Each work item contained 500 stripes of the first map to increase the granularity of computation.

The application obtains reasonable speedups for this data set up to 4 processors. After this point there is not enough work to adequately overcome the overhead of the parallelism due to the granularity of computation and the performance declines, as evidenced by the results for 8 processors.

Processors	1	2	4	8
Time (sec)	30	18	10	5
Speedup	-	1.67	3.00	6.00

Table 5.1: Speedups and wall clock times for overlaying two maps of 3000×3000 rectangles. The times given are the median of ten runs.

5.5 Summary

As opposed to the tradition parallelization of a pipeline where each stage is assigned one or more threads, the Pipeline pattern uses a work-pile approach in order to balance the load across the stages. The Pipeline pattern was used to implement a map overlay application. However, due to insufficient granularity in the problem, the application was unable to achieve good performance beyond 2 processors.

Chapter 6

The Wavefront Pattern

6.1 Overview of the Wavefront Pattern

The Wavefront pattern is applicable to applications where the data dependencies between work items can be expressed as a directed acyclic graph (DAG). The *wavefront* denotes the partition between nodes of the graph that have been computed and nodes that can now be computed because their dependency requirement have been satisfied. While a wavefront may occur in arbitrary DAGs (see Figure 6.1(a)), the Wavefront pattern restricts the set of dependency graphs to those which occur in a matrix (see Figure 6.1(b)). Parallelism in the Wavefront pattern results from elements on the wavefront being data independent of each other, otherwise the elements could not occur on the wavefront. Since the granularity of a computation for a single element in the matrix may be small, the granularity of a wavefront problem can be increased by grouping elements together while preserving the dependency relation between and within groups.

6.2 Chapter Overview

This chapter begins with an in-depth description of the Wavefront pattern in CO_2P_3S including the various parameters and an overview of the hook methods generated. How the Wavefront pattern is used in an application is also described. Section 6.4 describes the evolution of the Wavefront pattern. Section 6.5 gives an overview of the implementation of the pattern. Each of the following three sections describes how the Wavefront pattern was used to implement three different applications. Finally, a summary of the chapter is given.

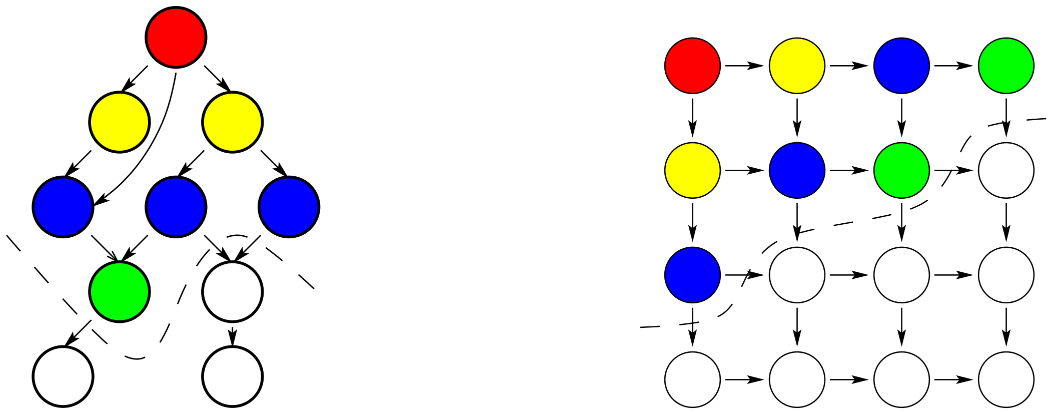
6.3 The Wavefront Pattern in CO_2P_3S

The Wavefront pattern is a new pattern in CO_2P_3S which was added as a result of the research described in this dissertation. It has been described in a previous publication [1]. The representation of the Wavefront pattern in CO_2P_3S is shown in Figure 6.2. This figure shows the default configuration for the pattern when it is selected in CO_2P_3S.

6.3.1 Pattern Parameters

The Wavefront pattern contains a single lexical parameter: the name of the class which represents a single element in the matrix. This class contains the hook methods that the CO_2P_3S user implements for their application.

The Wavefront pattern has three design parameters. Remember that design parameters have the greatest effect on the generated framework as they specify the overall structure of the framework. The design parameters for the pattern are:



(a) A wavefront in a dependency graph.

(b) A wavefront in a matrix.

Figure 6.1: Dependency graphs. Each colour represents a set of nodes which can be processed in parallel.

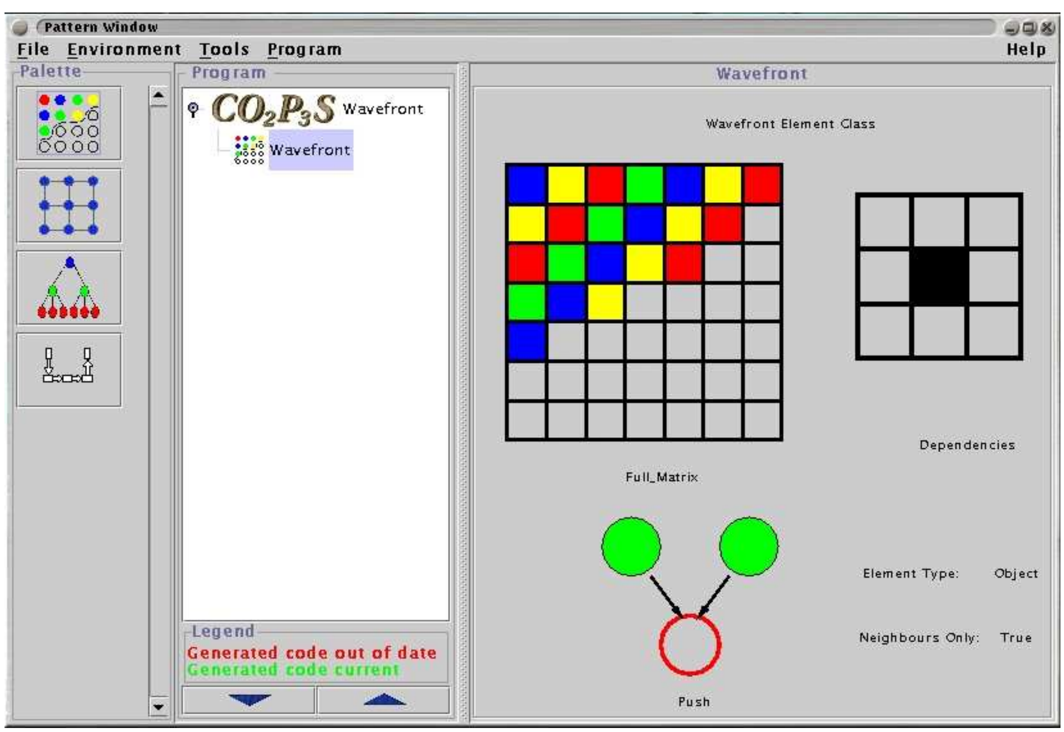
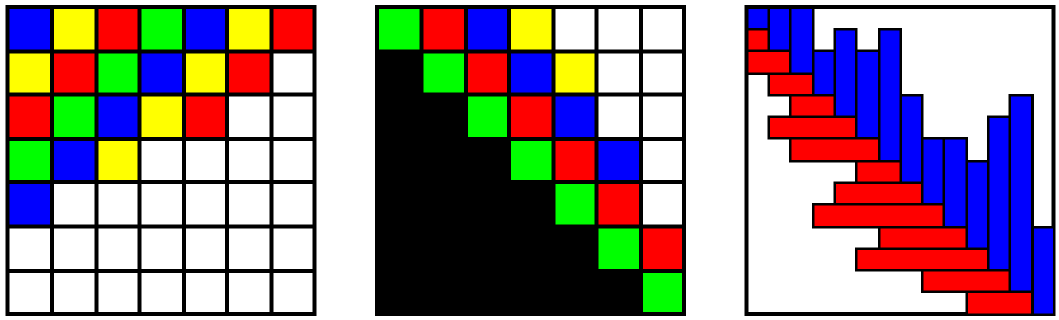


Figure 6.2: A screenshot of the Wavefront pattern in CO_2P_3S.



(a) A Full matrix shape. (b) A Triangular matrix shape. (c) A Banded matrix shape.

Figure 6.3: Matrix shapes in the Wavefront pattern.

1. *The shape of the matrix containing the elements.* The pattern supports three matrix shapes:

   Full where all elements of a rectangular matrix are computed (see Figure 6.3(a)),

   Triangular where only the elements in the top triangular portion are computed (see Figure 6.3(b)), and

   Banded where the values that are computed are centered around the diagonal (see Figure 6.3(c)).

2. *The dependency set for an item.* Dependencies are specified based on their direction relative to a matrix element. For example, an element may depend on elements that are north (N) and west (W). Not all sets of directions form legal dependency sets. An example of an illegal set is one containing opposing directions as this creates a cyclic dependency and would result in deadlock. The Wavefront Pattern GUI enforces these restrictions. All supported dependency sets contain directions which fall within a 90° arc.

3. *Whether an item needs access to more than its immediate neighbours for its computation.* Some applications require values which are not adjacent. For example, matrix multiplication requires access to all elements to the left and above a particular element.

The Wavefront pattern also has the following performance parameters. Recall that performance parameters tune an aspect of the generated framework:

1. *The notification method used to inform an item of the satisfaction of a dependency relation.* Items can use either the Pull notification method where the completion status of an item's prerequisites are polled, or the Push notification method where prerequisite items signal their dependents that they have completed their computation.

2. *The type of the item.* If the type of the element is a primitive, such as an `int`, then static hook methods are generated using the specific type. If the type specified is Object, instance hook methods are generated for the class defined by the user via the lexical parameter.

6.3.2 Hook Methods

The generated hook methods vary greatly and depend primarily on the value of the dependency set parameter, although the shape of the matrix also affects them. The Full matrix parameter value produces the most hook methods and the other matrix shapes generate a

```
operateCorner(int row, int column)
operateLeft(int row, int column, ...)
operateRight(int row, int column, ...)
operateTop(int row, int column, ...)
operateBottom(int row, int column, ...)
operateInterior(int row, int column, ...)
initialize(int row, int column, Object initializer)
reduce(int row, int column, Object reducer, ...)
```

Figure 6.4: The hook methods for the Wavefront pattern.

subset of them as appropriate. How the various methods are used is deferred until Section 6.5. The total range of methods produced are show in Figure 6.4.

Due to the shape of a Triangular matrix the `operateBottom()` method will never be produced and the `operateLeft()` method is used for the items on the diagonal which are not at a corner.

The `initialize()` and `reduce()` methods are always generated regardless of the parameter settings. The `initialize()` method is used for initializing all the items of the matrix before computation begins. Similarly, the `reduce()` method is used to collect information from the matrix. Each method takes as a parameter an object to be used in initialization and reduction respectively.

If the matrix type is Banded then two additional methods are also generated.

- `startRow(int column)` is called once for each column so that computation begins at the first row with a legal value for a particular column.
- `startColumn(int row)` is called once for each row so that computation begins at the first column with a legal value for a particular row.

The parameters of the `operate()` methods depend on the value of the neighbours-only access parameter. If the neighbours-only access is true, then the values for specific directions are provided such as `operateInterior(int row, int column, int north, int east)` if the element type was `int`. Otherwise a reference to the entire matrix is provided, such as `operateInterior(int row, int column, Matrix m)`, and the user accesses the necessary items by sending the `getElement(int row, int column)` message to the `Matrix` object.

The type parameter affects the type of template methods generated. If the parameter is of Object type then instance methods are generated, otherwise static methods are generated. For static methods the return type is the value of the type parameter, and for instance methods the return type is void. Also, the type of the parameters when the neighbours-only parameter is set to true are dictated by this value. If the type parameter is set to `int`, the method signature for `operateInterior()` with a {`N, E`} dependency set would be `static int operate(int row, int column, int north, int east)`. The same method would become `void operate(int row, int column, Element north, Element east)` if the element type were `Object` and lexical parameter was `Element`.

6.3.3 Using the Pattern

A Wavefront pattern is used in an application by the programmer creating a Wavefront object which acts as the controller for the threads. The constructor for a Wavefront takes as parameters the number of threads, the width and height of the matrix, an initializer object (or null if no object is required), a reduction object (or null if no object is required), and optionally an item blocking size. If no blocking size is provided then a default size (100 items by 100 items) is used. This value has been shown experimentally to generally produce blocks of elements of a sufficient granularity for good performance for the machines on which the Wavefront pattern was tested. In the future this value will need to be changed to accommodate improvements in technology. The Wavefront pattern assumes the responsibility for starting the threads, initializing the items, partitioning the items into groups, providing the threads with groups to process and executing the computations on each item without violating the specified dependency relations. To begin computation the programmer sends the `execute()` message to the created Wavefront object. To gather the results from the matrix, the user sends the `reduce()` method to the Wavefront object, which then calls the user-defined `reduce()` method on all the elements of the matrix.

6.4 Evolution of the Wavefront Pattern

The design of the Wavefront pattern went through several iterations before it reached its current state. Originally it was the responsibility of the user to create the blocks of elements to be processed and to specify the dependency relationships between them. The Wavefront user then created a controller object, passing it an array containing all the work blocks, an array of all the blocks with no dependencies (the initial set of blocks), and the number of threads. This original Wavefront pattern design contained only the *notification* parameter. While this design provided the most flexibility as any dependency graph could be constructed and used, it was found to place too much responsibility on the user and greatly increased the likelihood of errors, such as the introduction of cyclic dependencies, which violated the 'correctness' principle of CO_2P_3S. This preliminary version of the Wavefront pattern was used in the initial verification of the MetaCO_2P_3S tool [7].

As all the applications in this research that used the Wavefront pattern were matrix-based, the next version of the Wavefront moved the responsibility for creating blocks of elements and specifying the dependency graph into the framework in order to guarantee correctness. This was done at the cost of restricting the type of dependency graphs supported by the pattern to only those which can be specified by a matrix. This decision increased the complexity of the framework substantially, as the pattern needed to allow the user to specify the dependency set parametrically, of which there were found to be twenty-four unique legal sets. This led to the addition of the *dependency set* parameter.

This set of parameters was found to be sufficient to write the first wavefront application, a sequence alignment program.[1] However, when the second application (a skyline matrix solver) was tried, these parameters were found to be insufficient. First, as skyline matrices are a form of sparse matrices, the full matrix shape was inefficient as many of the elements were not used. This led to the addition of the *matrix shape* parameter. Further investigation of the matrix product chain solver application revealed that it too used a differently shaped matrix than the other two applications. This reinforced the need for the matrix shape parameter, and provided the third value for this parameter. Similarly, the skyline solver application led to the addition of the *neighbours-access* parameter. The sequence alignment application only required access to an element's immediate neighbouring values, whereas the skyline solver and matrix product chain solver required access to all elements along a row or column.

Once the design parameters were established and the applications implemented using the pattern, another deficiency in the pattern was discovered. It was found that the applications built using the Wavefront pattern provided poor performance both in terms of space and time. This was a result of the use of objects for representing the elements of the matrix. In all three applications the element type for an individual element was a primitive such as `int` or `double`, which was being wrapped in an object in order to work with the framework. As the performance of the applications was unacceptable, the *element type* performance

[1]This application is not one of the Cowichan Problems. It was selected prior to the decision to use the Cowichan Problems as an initial example of an application which used the Wavefront pattern.

1	2	3	4	5	7
2	3	4	5	7	9
3	4	5	7	9	11
4	5	7	9	11	12
6	8	9	11	12	13
8	10	11	12	13	14

(a) Diagonal synchronization.

1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	8	9
4	5	7	8	9	10
6	7	8	9	10	11
7	8	9	10	11	12

(b) Prerequisite synchronization.

Figure 6.5: Diagonal and prerequisite synchronization with 4 processors.

parameter was added to the pattern to remove this inefficiency from the pattern.

From this description of the evolution of the Wavefront pattern it appears that every new application results in a further evolution of a pattern. While this is true with a new pattern such as the Wavefront pattern, eventually the pattern will have matured enough to provide the necessary parameters to implement a wide variety of applications without the need for further evolution of the pattern. This trend can already be seen in the Wavefront pattern as the evolution of pattern between the skyline matrix solver and matrix product chain was only the addition of a third value to the existing matrix shape parameter.

6.5 Implementation of the Wavefront Pattern

The Wavefront pattern is implemented using a work queue model. As pieces of work become available though the satisfaction of their dependency constraints, they are placed onto a queue. Processors then request work from a controller object which manages the queue. The work queue implementation used in the Wavefront pattern is a modified version of Doug Lea's *LinkedQueue* [17] to avoid unnecessary synchronization and casting of objects.

There are two techniques which may be used to synchronize the workflow in a wavefront application: diagonal synchronization and prerequisite synchronization. In *diagonal synchronization*, an element is added to the work queue only if all the work from the previous diagonal has completed. In contrast, *prerequisite synchronization* only waits until the completion of an element's prerequisite constraints before adding the element to the work queue. Figures 6.5(a) and 6.5(b) respectively show the time steps in which elements would be processed using diagonal and prerequisite synchronization for 4 processors. As would be expected and as the figures show, diagonal synchronization is slower than prerequisite synchronization. However, as diagonal synchronization is deterministic with respect to time steps, it can be used to find the maximum theoretical speedup for the Wavefront pattern, assuming no other forms of overhead other than synchronization due to data dependencies. This can then be used to find the relative overhead of the pattern framework. Table 6.1 shows the actual and theoretical speedups for a version of a sequence alignment problem (Section 6.6) which uses diagonal synchronization. The difference between these two values gives an indication the relative overhead of the pattern framework. Considering how quickly that a wavefront application can be constructed, the overheads shown in Table 6.1 are fairly small.

Processors	Actual Speedup	Theoretical Speedup	Relative Overhead
2	1.66	1.99	17%
3	2.41	2.94	18%
4	3.04	3.88	22%

Table 6.1: Theoretical versus actual speedup for the Wavefront pattern framework.

```
initialize()

while(workDone < totalWork)
    work = queue.getWork()

    if(work is the corner)
        work.operateCorner()
    else if(work is on top edge)
        work.operateTop(row, column, int west)
    else if(work is on left edge)
        work.operateLeft(row, column, int north)
    else
        work.operateInterior(row, column, int north,
            int west)

    workDone++

this.reduce()
```

Figure 6.6: The `execute()` method.

Upon instantiation, the Wavefront object creates a matrix of the appropriate shape and populates it using the `initialize()` method. The matrix is then divided into work units based on the blocking factor (either the default or the user-specified one). The dependency graph of the work units is then constructed based on the dependency set and all units which have no dependencies are placed onto the work queue as the initial pieces of work. The system is now primed and awaits only the reception of the `execute()` method to create the threads and begin execution. Figure 6.6 shows the `execute()` method for a dependency set of {`N, W`} and element type of `int`. This code is unseen by the user.

The *Push* notification technique used in the Wavefront pattern is similar to the Observer pattern [12]. Since the pattern is parameterized by the dependency set, a work item knows the specific items in the matrix that are dependent on itself, and does not need to keep a collection of "observers" and can be more efficient by notifying the dependents directly. In the *Pull* notification technique when a work item finishes its computation, all of its dependents are collected and each is tested for "readiness". In testing the readiness of the dependents, each dependent queries the completion state of its prerequisites and states that it is ready when all of its prerequisites are completed. The collection of the dependents of a work item is an optimization to avoid the cost of having all queued items query their prerequisites when the majority of them will not yet be ready.

$$
\begin{aligned}
&DPM[row][column] = max(\\
&\quad DPM[row][column-1] - gapPenalty, \\
&\quad DPM[row-1][column] - gapPenalty, \\
&\quad DPM[row-1][column-1] + similarity(row, column))
\end{aligned}
\tag{6.1}
$$

6.6 The Sequence Alignment Problem

While this problem is not contained in the Cowichan Problem set, the sequence alignment problem started the project of finding additional patterns to add to CO_2P_3S. As such, it dictated much of the initial design of the Wavefront pattern and is included here.

6.6.1 Description of the Problem

In order to discover the function of certain genes or protein sequences, the alignment of sequences is common in computational biology. An alignment of sequences is typically done using a dynamic programming matrix (DPM) in conjunction with a similarity matrix for determining the score for matching certain elements. As part of the alignment, gaps may be inserted into either of the sequences to produce a better alignment, though there is typically a penalty associated with this action. Table 6.2(a) shows a dynamic programming matrix for the two sequences TLDKLLKD and TDVLKAD, and Table 6.2(b) gives an example of a similarity matrix. A particular value of the DPM is dependent on the values from the left (west), above (north), and upper-left (northwest) cells. Therefore the computation in such a DPM proceeds from the top left-hand corner to the lower right-hand corner. The values of cells in the DPM are found using Formula 6.1. The gap penalty used for constructing Table 6.2(a) is a constant (10), though it could also depend on the number of gaps previously inserted at that location so that large gaps are discouraged. Once all the values of the DPM are computed, a path is traced from the lower right-hand corner back to the upper left-hand corner to produce the alignment of the two sequences.

6.6.2 Why the Wavefront Pattern is Appropriate

In general, the computation of a matrix element in the DPM depends directly on the values above, to the left and upper-left diagonal.[2] Therefore the computation of the elements along one diagonal are strictly dependent on the values of the previous diagonal. This dependency relationship between elements makes the Wavefront pattern ideal for this problem.

6.6.3 Using the Wavefront Pattern

The design parameter settings for the sequence alignment problem are matrix type Full Matrix, dependency set {N, W, NW}, and neighbours-only true as shown in Figure 6.7. The two performance parameters are set to `int` element type and Push notification. This parameter configuration generates four template methods: `operateInterior()`, `operateLeft()`, `operateTop()`, and `operateCorner()`. The `operateCorner()` method returns the value 0, and the `operateLeft()` and `operateTop()` methods return the fixed values of the left column and top row respectively. Alternatively, the `initialize()` method could set all the values for the top row and left column and the `operateCorner()`, `operateLeft()`, and `operateTop()` methods would simply return the value at a particular location. This is a design decision of the application developer and does not affect how the pattern behaves. The `reduce()` method collects the value of the lower-right corner as representing the optimal

[2] The only exceptions are for the top row and left column where some of these values do not exist.

	-	T	L	D	K	L	L	K	D
-	0	- 10	-20	-30	-40	-50	-60	-70	-80
T	-10	20	10	0	-10	-20	-30	-40	-50
D	-20	10	20	30	20	10	0	-10	-20
V	-30	0	22	20	30	32	22	12	2
L	-40	-10	20	22	20	50	52	42	32
K	-50	-20	10	20	42	40	50	72	62
A	-60	-30	0	10	32	42	40	62	72
D	-70	-40	-10	20	22	32	42	52	82

(a) Dynamic programming matrix for the two sequences.

	Name	A	D	K	L	T	V
A	alanine	16	0	0	0	0	0
D	aspartic acid	0	20	0	0	0	0
K	lysine	0	0	20	0	0	0
L	leucine	0	0	0	20	0	12
T	threonine	0	0	0	0	20	0
V	valine	0	0	0	12	0	20

(b) Similarity matrix for the two sequences.

Table 6.2: Dynamic programming and similarity matrices for the alignment of sequences TLDKLLKD and TDVLKAD.

alignment score for the two sequences. Typically such a program would then trace a path from the bottom right-hand corner to the top left-hand corner to extract the alignment. However, since this is a sequential process, it was not part of this application as the focus of the application was on the parallelism and behaviour of the Wavefront pattern.

6.6.4 Results

Using portions from a sequential version of the application, the parallel version was written using CO_2P_3S in about an hour. The sequential application contained 5 classes totaling 133 lines of code. The parallel application consisted of 15 classes, 9 of which were generated and 4 of which were reused from the sequential version. The driver program for the sequential application was modified to use a Wavefront object in the parallel application, with better than half of the original code being reused. Of the 358 lines of code in the parallel version (235 of which was framework code), the user was required to write 28 lines of new code.

The program was run using two sequences of length 10,000 using the default blocking strategy (100 blocks by 100 blocks) on an SGI Origin 2000 with 46 MIPS R100 195 MHz processors and 11.75 gigabytes of memory. The program was run on a native threaded Java implementation from SGI (Java 1.3.1) with optimizations and JIT turned on. The virtual machine was started with 1 GB of heap space. Table 6.3 shows the results of taking the median of 10 runs of the program. The times exclude initialization, reduction, and output and represent only computation time. As can be seen, the application obtains good speedups up to 16 processors. When 32 processors are used, there is a marked decline in performance, but this is to be expected. First, the matrix is divided into 100 x 100 blocks which leaves many processors idle at the beginning and end of the computation. At 32

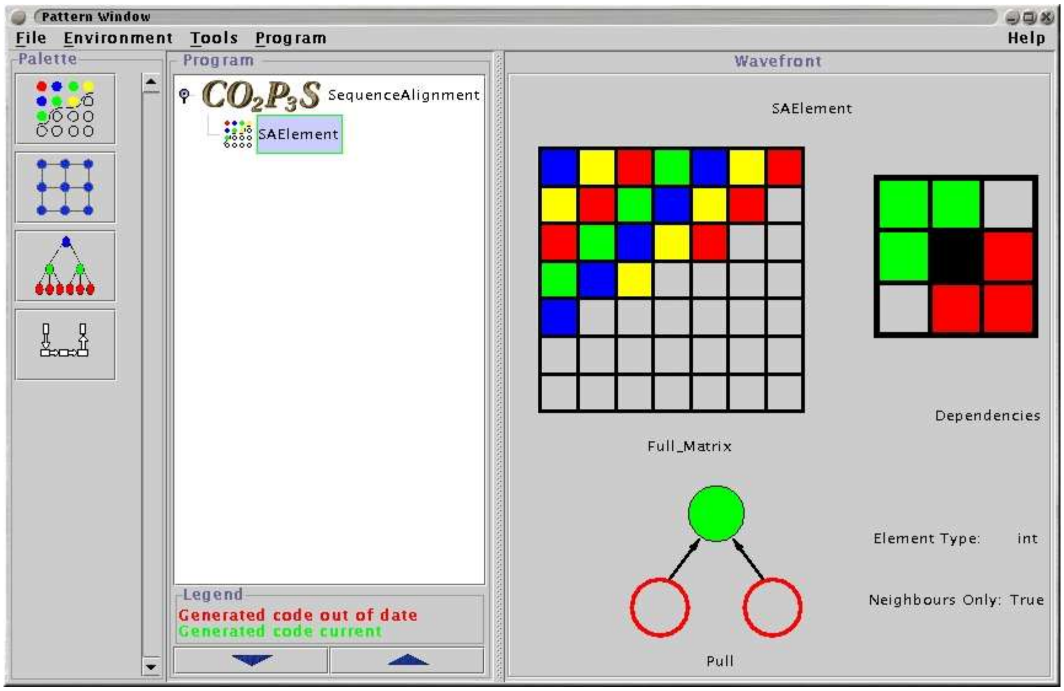


Figure 6.7: Screenshot of Wavefront pattern parameterized for sequence alignment application.

processors less than 32 processors are being utilized for 11% of the time, as compared to 1% of the time for 16 processors. The blocking factor can be changed to minimize the number of the idle processors, but this will decrease the granularity of the blocks and adversely affect the performance in this manner. Also, the processing time for the application is so low for 16 processors that the overhead of the framework may be affecting the time for 32 processors. Regardless, it is obvious that there is not enough work generated to support 32 processors. Larger sequences could not be used due to memory constraints imposed by the JVM.

Processors	1	2	4	8	16	32
Time (sec)	69	37	19	10	5	4
Speedup	-	1.89	3.65	7.05	13.32	18.15

Table 6.3: Speedups and wall clock times for aligning two sequences of length 10,000.

6.7 A Skyline Matrix Solver

6.7.1 Description of the Problem

A skyline matrix is an NxN matrix in which each row has an indentation up to the diagonal and each column has a similar indentation, as shown in Figure 6.8. More formally, there exists constants r_i and c_i such that $1 \leq r_i \leq i$ and each row i has non-zero values from r_i to i, and $1 \leq c_j \leq j$ and the column j has non-zero values from c_j to j [31]. While any matrix

21	45	0	0	0	0	0	0	0	0
0	85	27	0	7	0	0	0	0	0
0	0	25	34	11	0	0	0	0	0
0	0	91	2	17	0	0	0	0	0
0	3	19	3	29	32	0	82	0	0
0	0	74	42	10	61	0	25	0	37
0	0	0	66	8	18	12	50	0	85
0	0	0	48	7	28	3	54	0	25
25	7	29	17	88	47	9	30	2	29
0	0	0	0	0	0	0	90	67	1

Figure 6.8: Example of a 10x10 skyline matrix with indentation vectors.

may be viewed as skyline matrix,[3] only skyline matrices with a substantial zero-element content are of interest, i.e. matrices with values clustered around the diagonal. The name for this type of a sparse matrix is derived from its similarity in shape to a city skyline.

Given a skyline matrix A and a solution vector b, we want to solve $Ax = b$ using LU-decomposition. What makes this problem interesting is that many of the matrix elements are zero resulting in many of the inner products being zero. The key to efficiently solving a skyline matrix is to exploit this property. Doolittle's method of LU-decomposition is therefore used to compute the inner products [5]. The value of a matrix element a_{ij} is found by Formula 6.2 if it is in the upper triangular portion and Formula 6.3 if it is in the lower triangular portion.

$$u_{ij} = (a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}) \quad \text{for } 1 \leq j \leq n, 1 \leq i \leq j \tag{6.2}$$

$$l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{ij}) / u_{jj} \quad \text{for } 1 \leq i \leq n, 1 \leq j \leq i \tag{6.3}$$

6.7.2 Why the Wavefront Pattern is Appropriate

The dependency relation between matrix elements dictates the use of the Wavefront pattern. Figure 6.9 shows the elements which are required to compute either `U` in the upper triangular portion, or `L` in the lower triangular portion. Elements in the upper triangular portion are dependent on all elements of the same row in the lower triangular portion and all elements directly above (i.e. in the same column). Elements in the lower triangular portion are dependent on all elements directly to the left of the element (i.e. the same row) and all elements in the same column that occur in the upper triangular portion, including the diagonal element. In other words, `U` can be computed once the values above and to the left are known, and similarly `L` can be computed once the values to the left and on the diagonal are known.

[3] A full matrix is a skyline matrix which has a zero indentation for all rows and columns.

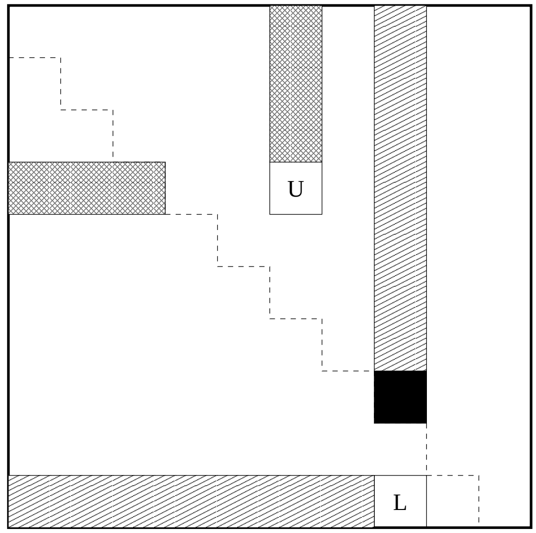


Figure 6.9: Dependency relations for skyline matrix using Doolittle's method.

6.7.3 Using the Wavefront Pattern

The parameter settings for this application are as follows: the dependency set is {N, W}, the matrix shape is Banded, and the neighbours-only flag is set to false as shown in Figure 6.10. The performance parameters are set to `double` element type and Push notification. Due to the dependency set, the methods `operateInterior()`, `operateLeft()`, `operateTop()`, and `operateCorner()` are all generated.

6.7.4 Results

Taking a sequential version of the program, parallelization of the program using CO_2P_3S took a few hours. The sequential version of the application contained 2 classes, both of which were reused in the parallel application, totaling 196 lines of code. The class for constructing and processing the matrix was subclassed in the parallel version and the appropriate method was overridden to use a Wavefront object instead. The 15 lines in the sequential application which performed the calculation for a single element were reused for the `operateInterior()` hook method. In total, the parallel application reused 144 lines of code from the sequential application. The parallel application consisted of 12 classes, of which 9 were the framework classes. The application totaled 390 lines with 224 being generated by CO_2P_3S. The user was required to write 22 new lines of code for the parallel application.

The program was run on a 50% full skyline matrix of size 2000 $\times$ 2000 with a blocking factor of 100 blocks by 100 blocks (the default). The machine used was the same SGI Origin 2000 which was used for the sequence alignment problem (Section 6.6), and the same run-time environment was used (native threads, JIT turned on, virtual machine started with 1 GB heap space). Table 6.4 shows the median of 10 runs and that this application also achieves good speedups up to 16 processors. Again the times given are strictly the computational times and do not include the time for initialization, reductions, or output. As was previously seen, the speedups fall off towards 32 processors due to many processors being idle near the start and end of the computation.

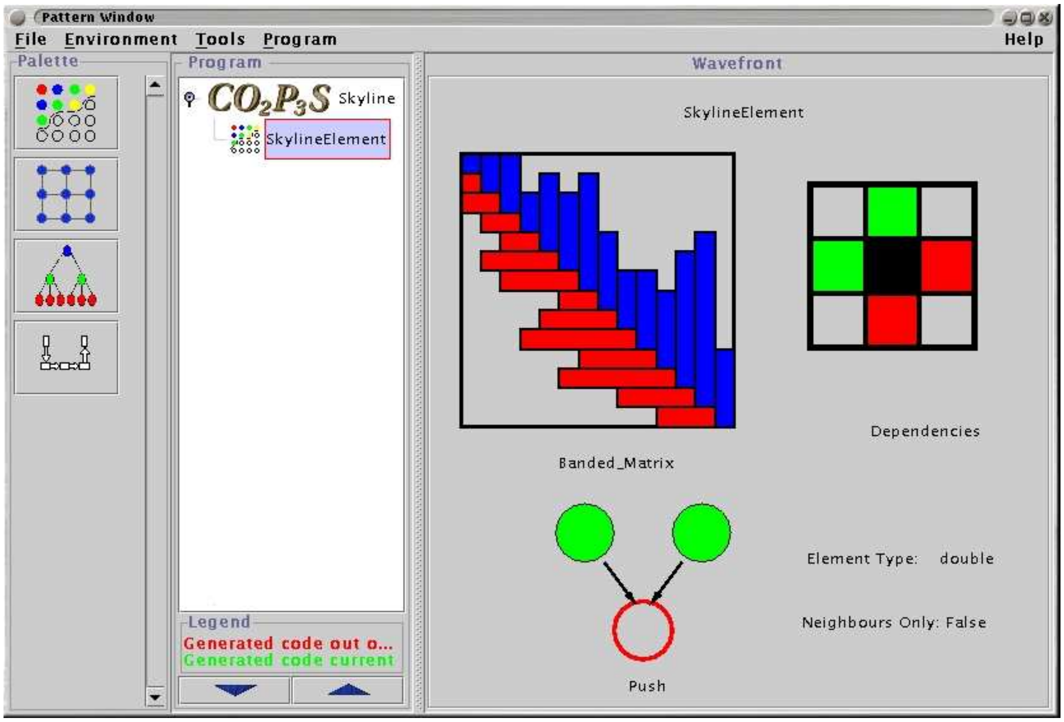


Figure 6.10: Screenshot of Wavefront pattern for a skyline solver application.

Processors	1	2	4	8	16	32
Time (sec)	446	232	115	57	30	26
Speedup	-	1.93	3.89	7.84	14.86	17.15

Table 6.4: Speedups and wall clock times for decomposing a 50% full 2000 $\times$ 2000 skyline matrix.

6.8 Solving Matrix Product Chains

6.8.1 Description of the Problem

Given a program that must multiply a series of M_i matrices for $0 \leq i \leq N$ and each matrix has r_i rows and c_j columns, what is the order in which the matrix multiplications should be done such that the least number of scalar multiplications is done? The formula $r_i c_i c_{i+1}$ gives the number of scalar multiplications performed to find the product of two matrices. Given the matrices A, B, C, D with dimensions 30 $\times$ 17, 17 $\times$ 12, 12 $\times$ 23, and 23 $\times$ 16 the number of scaler multiplications can range from 25,440 for $((AB)C)D$ to 15,840 for $A(B(CD))$.

The solution to this problem is found by filling the upper triangular portion of a dynamic programming matrix C with the minimum cost of finding the matrix product for matrices M_i and M_j[31]. Table 6.5 shows an example. The minimum cost is found by determining the least cost for finding the products $M_i \ldots M_k$ and $M_{k+1} \ldots M_j$ and using these to find the cost of the product for $M_{i \rightarrow k}$ and $M_{k+1 \rightarrow j}$. Once the upper portion is filled, the least cost will be the value in the top right-hand corner.

	A	B	C	D
A	0	6120	14400	15840
B	0	0	4692	7680
C	0	0	0	4416
D	0	0	0	0

Table 6.5: An example of a dynamic programming matrix for the matrix chain product problem.

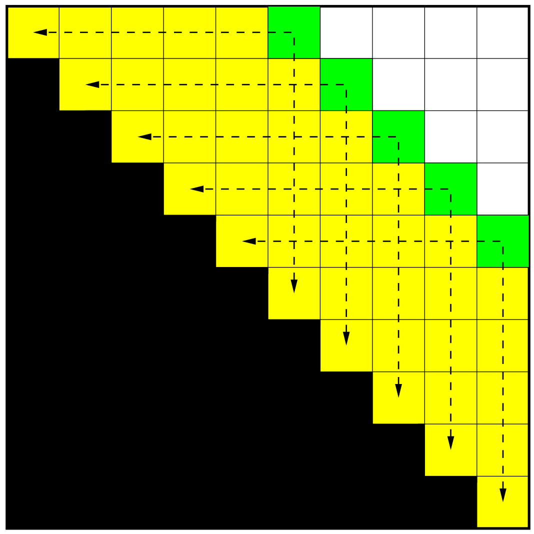


Figure 6.11: The dynamic programming matrix for the matrix product chain problem can be solved on a diagonal-by-diagonal basis.

6.8.2 Why the Wavefront Pattern is Appropriate

The Wavefront pattern is appropriate for solving this problem due to the dependency relation between matrix cells in the problem. If the values of $C_{i,k}$ and $C_{k+1,j}$ are known, then the calculation of $C_{i,j}$ can proceed. Therefore, if the values of the matrix are found on a diagonal-by-diagonal basis [31], then the values of the next diagonal can be found independently of each other as shown in Figure 6.11. The values can be found in a wavefront manner due to the {S, W} dependency relationship between matrix cells.

6.8.3 Using the Wavefront Pattern

For the Matrix Product Chain the dependency set is {S,W}, the matrix type is Triangular, and neighbours-only is false as shown in Figure 6.12. As with the sequence alignment problem (Section 6.6), the performance parameters are set to Push notification and `int` element type. As the dependency set is {S, W}, the only template methods generated are `operateCorner()` and `operateInterior()`. Unlike the Sequence Alignment and Skyline Solver problems which each had an initial working set of size 1 (the upper left-hand corner), the initial working set for the Matrix Product Chain is all the elements along the diagonal of the matrix. However, this is internal to the framework and the user of the pattern is never directly aware of this difference.

For this problem the `initialize()` method simply sets all the costs to 0 along the

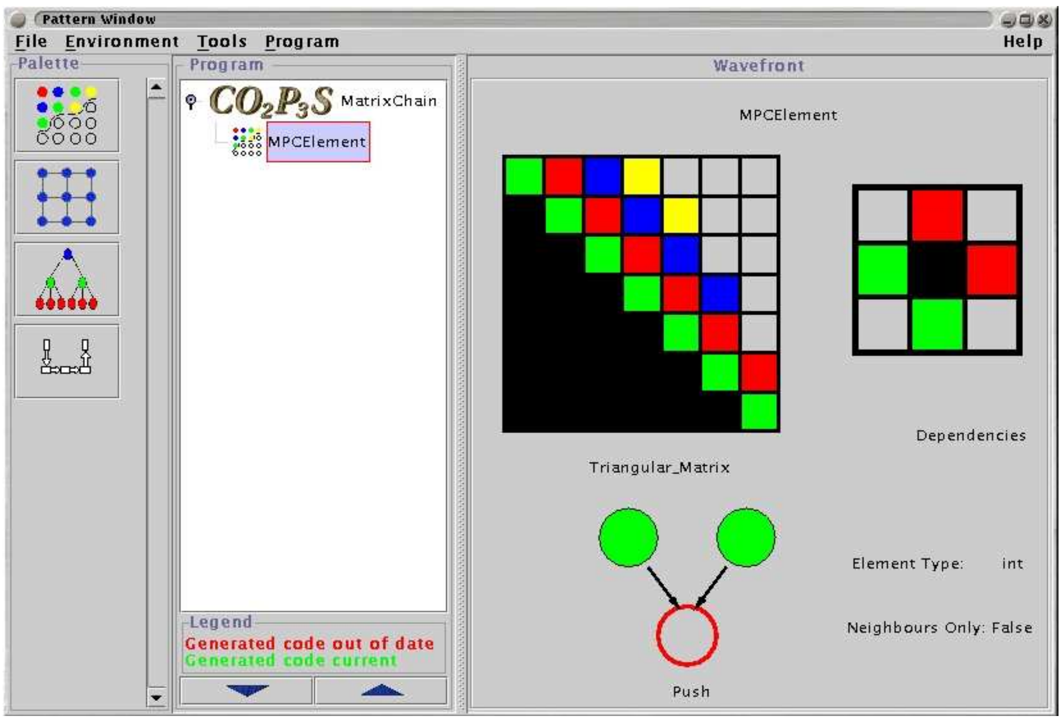


Figure 6.12: A screenshot of the Wavefront pattern parameterized for matrix product chain application.

```
minCost = Integer.MAX_VALUE;

for(k = row to column-1) do
    cost = aMatrix.getElement(row,k)
        + aMatrix.getElement(k+1,column)
        + dimension[k+1] + dimension[column+1]
    if(cost < minCost)
        minCost = cost
end do

return minCost;
```

Figure 6.13: The `operateInterior()` method for the matrix product chain problem.

diagonal and all the other matrix cells to a maximum value. The `reduce()` method rejects the values of all cells and only stores the value of the top-right corner element, as this is the only value of interest. The `operateInterior()` method uses an application-specific array `dimension[]` and is shown in Figure 6.13.

6.8.4 Results

Similar to the previous two problems, given an existing sequential version of the application, it took only a small effort to generate the framework. The sequential version consisted of 68 lines of code over 2 classes, one of which was reused in the parallel version. The parallel application contained 12 classes, 9 of which were generated. Of the 296 lines of code in the parallel application, 223 lines were from the generated framework and 60 were reused from the sequential version. The user was required to add 13 new lines of code.

The program was used to find the minimum cost for multiplying 2,000 matrices of random dimensions between 10 and 100. The blocking factor used was the default 100 blocks by 100 blocks. The hardware/software configuration was the same as the previous two problems. Once again, Table 6.6 shows the median of 10 runs and that the application obtains good speedups up to 16 processors.The time taken by initialization, reduction, and output is not included in the times. As with the previous two applications many processors are idle near the beginning and end of the computation due to the blocking factor and performance is adversely affected for 32 processors.

Processors	1	2	4	8	16	32
Time (sec)	896	494	246	121	67	49
Speedup	-	1.81	3.64	7.40	13.37	18.29

Table 6.6: Speedups and wall clock times for solving the matrix product chain for 2000 matrices of random dimensions.

6.9 Summary

The Wavefront pattern is used to parallelize matrix applications in which the data dependencies can be expressed as a directed acyclic graph. Dynamic programming matrix applications are good examples of these types of applications. It is a new pattern to CO_2P_3S and was used to implement three applications: a sequence alignment application from computational biology, a skyline matrix solver, and a matrix product chain solver. Good speedups for up to 16 processors were achieved for all three applications.

Chapter 7

The Search-Tree Pattern

7.1 Overview of the Search-Tree Pattern

The Search-Tree pattern is used to parallelize tree search algorithms. Tree search algorithms are commonly used in areas such as optimization and heuristic search. The nodes of the tree represent states (e.g. a game board configuration) and the arcs represent movement between states (e.g. a player's move). The Search-Tree pattern uses the divide-and-conquer technique for searching a tree in which the children of tree nodes are generated up to a certain depth in the tree (divide) and the remaining nodes are processed sequentially by the processor (conquer). Figure 7.1(a) shows an example of this technique.

7.2 Chapter Overview

This chapter begins by describing the Search-Tree pattern in CO_2P_3S. It then proceeds with descriptions of two Cowichan Problems that were solved using this pattern; the Fifteen Puzzle in Section 7.6 and the Game of Kece in Section 7.7. Finally, some concluding remarks are made.

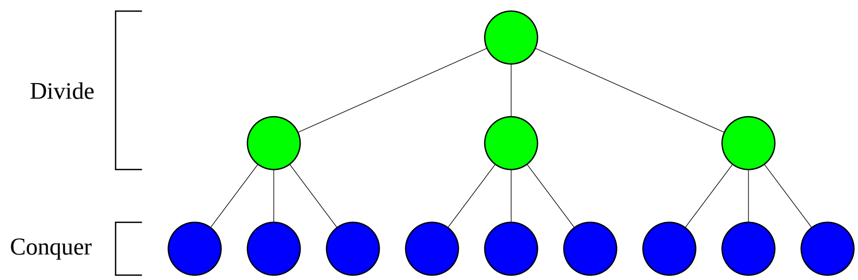
7.3 The Search-Tree Pattern in CO_2P_3S

The Search-Tree pattern is a new pattern in CO_2P_3S and was added as a result of this research in order to implement the two problems. A screenshot of the Search-Tree pattern in CO_2P_3S is shown in Figure 7.2.

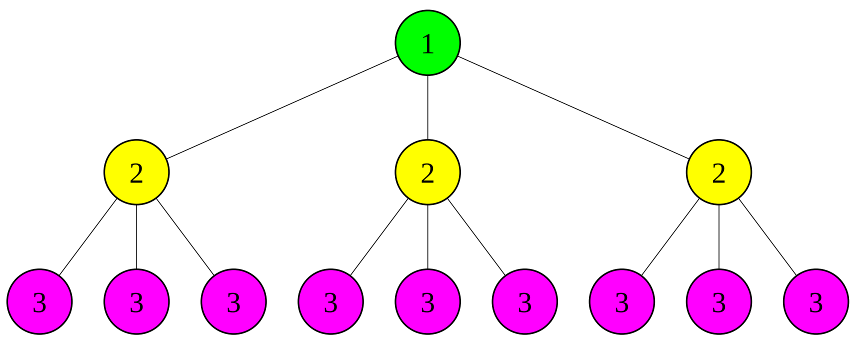
7.3.1 Pattern Parameters

The single lexical parameter for the pattern is the name of the class which represents a node in the tree. Recall that lexical parameters are place holders for class or method names in the framework. This class will contain the hook methods which are implemented by the CO_2P_3S user.

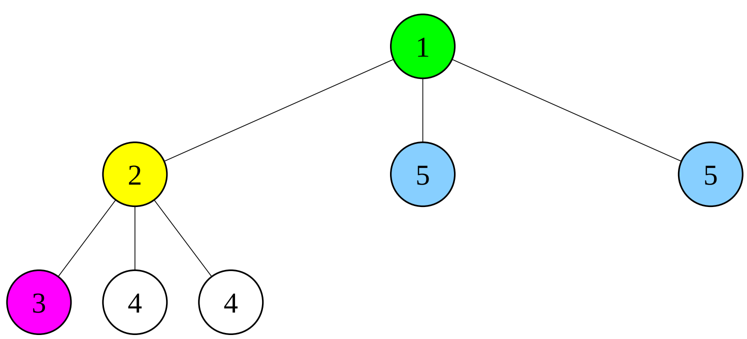
The pattern has a single design parameter, the *traversal technique*. The tree can be searched in either a breadth-first or a depth-first manner. If the tree is searched breadth-first then all nodes to a certain depth are expanded in parallel and the remaining children are then searched in parallel. Figure 7.1(b) shows the order in which nodes are processed for a breadth-first parallel tree search. If the tree is searched depth-first then all nodes on the left side of the tree are expanded to a certain depth and the left child at the specified depth is searched sequentially. Once a left child completes its computation, the sibling nodes are processed in parallel. Figure 7.1(c) shows the order in which nodes are processed for a depth-first parallel search.



(a) A divide-and-conquer tree.



(b) Breadth-first traversal.



(c) Depth-first traversal.

Figure 7.1: Tree traversals in the Search-Tree pattern. Nodes with the same value are processed in parallel.

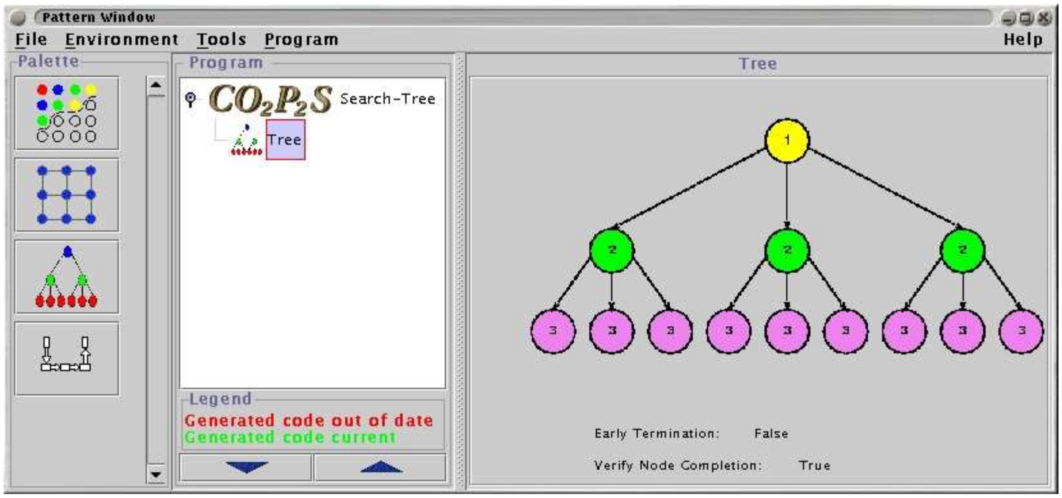


Figure 7.2: The Search-Tree pattern in CO_2P_3S.

The Search-Tree pattern has a single performance parameter, *early termination*. The early termination parameter allows for the termination of the search to occur before all nodes have been searched. For example, an application might want to terminate after finding one solution. Without early termination all solutions would be found. If the *early termination* parameter is selected then two framework methods are generated to allow the user to terminate the search before all nodes are searched. The first is `canContinue()` which is placed in the body of the hook method `conquer()` (see Section 7.3.2). The `canContinue()` method will return `true` as long as the user has not asked for the search to terminate. The user terminates a search by calling the other generated framework method, `terminateAll()`, which requests that all threads searching the tree be terminated.

The Search-Tree pattern introduces a new type of parameter to the CO_2P_3S system called a *verification* parameter. This type of parameter allows the pattern designer to include code in the framework to ensure its proper use and to find faults in the code of the user. In the Search-Tree pattern, the verification parameter is the ***done()* verification** parameter which verifies that the user's `done()` method is valid. The `done()` method is a hook method in which the user indicates when a node has completed its computation. If the user states that a node has not finished its computation, but CO_2P_3S detects that in fact it has, this indicates a fault in the user's code and an exception is thrown.

7.3.2 Hook Methods

The hook methods generated in the Search-Tree pattern do not depend on the parameters settings. Regardless of the parameter values, the same methods are always generated. How these methods are used in the pattern is deferred until Section 7.5. The hook methods are:

`parallel()` This method indicates whether to generate a node's children (i.e. call `divide()`) or proceed with computation of the node (i.e. call `conquer()`). A default implementation which returns true if the node is less than a specified depth is provided.

`divide()` This method generates a node's children. The children are returned as an array of tree nodes.

`conquer()` This method performs the sequential computation of a node.

`updateState(TreeNode child)` This method allows a node to update its state based on information which may be extracted from the child. When each child has completed its computation, it sends this message to its parent with itself as an argument.

`done()` This method specifies when a node is considered to be finished, such as when all children have updated their parent. This is the default behaviour.

7.3.3 Using the Pattern

The Search-Tree pattern is used in an application by creating a Tree object and then sending it the `traverse()` message. The Tree object acts as a controller for the threads and provides threads with nodes to process from a queue. To create a Tree object, first the root TreeNode of the tree must be created and passed to the Tree constructor along with the number of processors to use. Optionally, the user may specify the depth to which children are generated. If no depth is provided, then a default depth of 4 is used. The traversal of the tree is completed when the `traverse()` method returns.

7.4 Evolution of the Search-Tree Pattern

The Search-Tree pattern was created by developing an initial framework consisting of the `parallel()`, `divide()`, and `conquer()` methods. The Kece and Fifteen Puzzle problems were then implemented using this framework. Once these two applications were created, their implementations were compared to discover the parameters for the pattern. The traversal method was an obvious difference between the applications. Since any solution, as opposed to all solutions, is sought in the Fifteen Puzzle, the need for an *early termination* parameter was clear. As the Kece problem uses an alpha-beta search, which requires data to be propagated back up the tree, the need for the `updateState()` method was noted. In order that ancestors of nodes might be updated at the correct time, the `done()` method was added to indicate when a node was completed. As the framework evolved, it became evident that verifying the user's `done()` method could easily be accomplished, and the verification parameter was added to the pattern.

Adding the Search-Tree pattern using MetaCO$_2$P$_3$S was straight-forward as the parameters for the pattern are simple. Most of the work required to add the pattern involved separating the method bodies from the method signatures and then annotating the signatures and bodies. Given that the framework code was already written in the form of the two applications, it took only nine hours to enter the pattern using MetaCO$_2$P$_3$S and verify it with the two applications. This time is strictly how long it took to take existing framework code from the hand-generated Kece and Fifteen Puzzle programs and create the pattern using MetaCO$_2$P$_3$S. This does not include the time spent in designing the Search-Tree Pattern or writing the hand-coded versions of the applications.

7.5 Implementation of the Search-Tree Pattern

The Search-Tree Pattern uses a work queue model similar to that of the Wavefront pattern for managing the nodes of the tree. When a node is divided its children are placed on the queue and a fixed number of threads are fed work from that queue. As with the Wavefront pattern, a modified version of Doug Lea's *LinkedQueue* [17] was used. Computation of a node is accomplished via the `process()` method shown in Figure 7.3. In the case of depth-first traversal, a second queue (a pending queue) is used to hold the siblings of a left child until it has been processed. For a depth-first search, when the children are returned from `divide()` the first node in the array is assumed to be the left child and is placed immediately into the work queue. The remaining children are marked to indicate that they are dependent on the left child node and placed onto the pending queue. When a node

```
if(node is invalid) return

if(parallel())
    children = divide()

    /**
        This code will only appear if the
        breadth-first parameter setting is selected.
    */

    if(breadth-first traversal)
        foreach child
            add child to work queue

    /**
        This code will only appear if the
        depth-first parameter setting is selected.
    */

    if(depth-first traversal)
        mark first child as left child
        add left child to queue
        foreach remaining child
            add to pending queue
else
    conquer()
    update parent
```

Figure 7.3: The `process()` method.

has completed processing, the pending queue is searched for all nodes which depend on the completed node and if any are found they are placed onto the work queue. Once a node has completed processing, all the children of the node are marked as invalid in case there was an early termination and there were still nodes in the work queue to be processed. Processing of an invalid node returns immediately as shown in Figure 7.3.

Note that while there are many tools which could produce code like that shown in Figure 7.3, CO_2P_3S uses generative design patterns and either the portions which relate to breadth-first traversal or the portions for depth-first traversal would be generated. Once a traversal method has been selected, the opposing portions would not be generated, including the test for traversal type. This is an example of how generative design patterns can improve the performance of framework code.

The verification of the `done()` method is accomplished in the following manner. When `divide()` returns the children of a node, the children are all placed in a separate list used for keeping track of which children have finished. As each child finishes and updates their parent, the respective child node is removed from the list. Every time that `done()` returns `false`, the list is checked to see that it is non-empty. If the list is ever empty (i.e. `verifyDone()` returns `true`), when `done()` returns `false`, then an error has occurred since there are no more children that require processing and the current node must be finished. Figure 7.4 shows how updates are propagated up the tree and node completion is verified.

```
update state
remove child from validation list

/**
    This code will only appear if the
    depth-first parameter setting is selected.
*/

if(depth-first traversal)
    add nodes to work queue nodes from the pending queue
        which are now ready

if(done())
    invalidate all children
    update parent
else if(verifyDone())
    throw exception
```

Figure 7.4: The `update()` method.

7.6 The Fifteen Puzzle

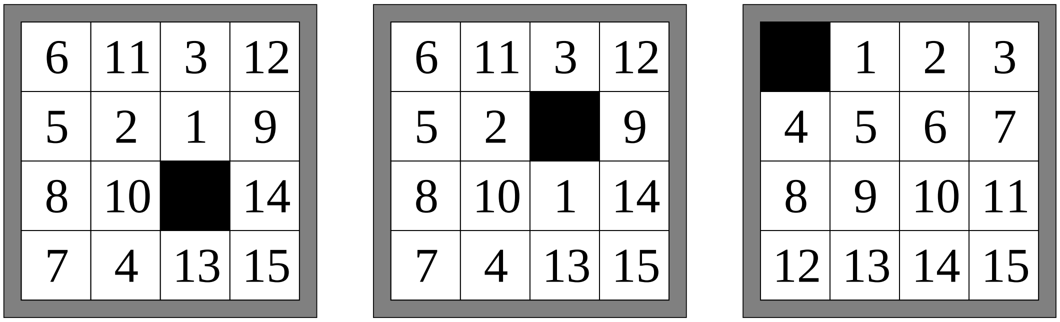
The Cowichan Problems contain a single-agent search problem, the Active Chart Parsing Problem. The problem involves generating all possible derivations of a sentence based on an ambiguous grammar. Unfortunately, finding grammars and sentences sufficiently large to produce programs which run for more than a few seconds on current processors is difficult. The sequential solution to the problem in the Orca study [27] from 1995 used a sentence of 29 words and ran in 27 seconds. This same program today would take only a few seconds to run. Also, the proportion of communication between the processes compared to the amount of computation is high (i.e. very fine granularity). This is the primary reason that the results of the Orca study showed that parallelism actually slowed down the computation. Therefore, a different single-agent search which was more representative of current problems was selected for this research. The single-agent search problem of solving the Fifteen Puzzle was selected as the replacement.

7.6.1 Description of the Problem

Of the sliding tile puzzles, the Fifteen Puzzle is the one most frequently used by researchers. The puzzle consists of a 4 x 4 board with an arrangement of fifteen numbered tiles and one blank space as shown in Figure 7.5(a). The object of the puzzle is to move tiles into the blank space (Figure 7.5(b)) in order to place all the tiles into numerical order (Figure 7.5(c)) and do so in the least number of moves.

To solve this problem a variant of A* search, depth-first iterative A* (IDA*) search, is used [23]. In A* search, a list is kept of nodes to be explored and they are considered in order of most to least promising. The "promise" of a node is based on the heuristic value of a node given by $f(n) = g(n) + h(n)$ where $g(x)$ is the cost of the path to a specific node from the root node and $h(x)$ is an estimate of how much farther it is to the goal. As long as $h(x)$ is *admissible* (i.e. never overestimates the distance to the goal), then A* is guaranteed to find the optimal path to the solution. However, the processing time and memory cost of maintaining the priority list of nodes to explore (and a list of nodes already explored to avoid duplicate searches) can be prohibitive.

IDA* addresses these problems by performing a depth-first search on iteratively larger



(a) An unsolved 15-puzzle.

(b) Moving a tile in the 15-puzzle.

(c) A solved 15-puzzle.

Figure 7.5: The Fifteen Puzzle.

trees. The search is controlled by a threshold value t, which is initially set to the $h(x)$ for the root node. The tree is then searched until either the goal is found or $f(x) > t$ for all branches of the tree. If the goal is not found in the current search then t is increased and the search is performed again with the new value of t. Figure 7.6 shows an example of using IDA* to find a solution. This process of iteratively deepening the tree until a solution is found allows for a reduced memory requirement as only the nodes for the current depth-first search need be stored at the cost of repeated computation.

For the Fifteen Puzzle, $g(x)$ is the depth of a node in the tree and the sum of Manhattan distances is used for $h(x)$. The Manhattan distance for a puzzle tile is the sum of the horizontal and vertical distances from the location of a tile to its position in the solution.

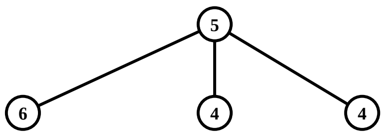
7.6.2 Why the Search-Tree Pattern is Appropriate

As this problem uses a single-agent search algorithm for finding the solution, the Search-Tree pattern is applicable to the problem.

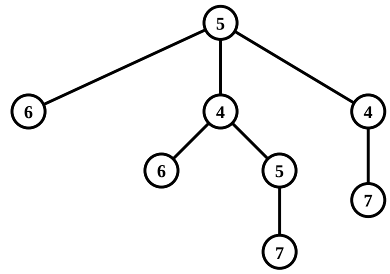
7.6.3 Using the Search-Tree Pattern

For this problem we would like to perform an IDA* search in parallel on multiple branches of the tree. Therefore the *traversal* parameter is set to breadth-first. Note that while the dividing of nodes in done in a breadth-first fashion, the children are sequentially searched in a depth-first fashion. As soon as a solution is found, the search should terminate, so the *early termination* parameter is set to `true`. Figure 7.7 shows the parameter settings for the problem in CO_2P_3S.

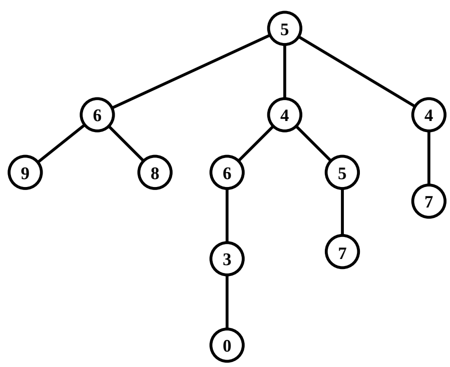
The default implementation of the `parallel()` method which expands nodes to a certain depth was used. The `divide()` method creates nodes for all possible movements of tiles from a given board. The `conquer()` method encapsulates a call to a recursive sequential method which performs the IDA* search for a subtree. If the method indicates that a goal has been found, then `terminateAll()` is called. When a node has been updated, if a child node reports that it contains a subtree with the goal node in it, then `updateState()` places the node on to a list so that the path toward the goal is remembered. The entire path to the goal is not saved, only up to the `conquer()` node which leads to a goal, as `updateState()` is only called on nodes which have been generated by `divide()`. The `done()` verification parameter was used during testing of the program, but was turned off for benchmarking.



(a) The initial search threshold is $h(s)$ where s is the root of the tree, 3 in this case. No solution is found for this threshold.



(b) The threshold is incremented to 5. The tree is now searched using this new threshold, and again no solution is found.



(c) The solution is finally found when the threshold is 7. As a solution was found, the branches on the right-hand side are not searched, even though they may also lead to a solution.

Figure 7.6: An example of IDA*.

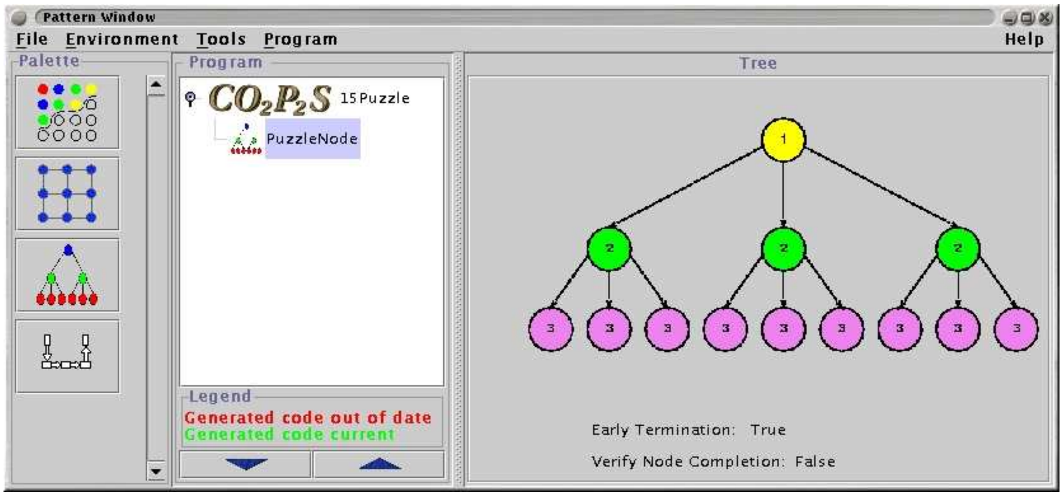


Figure 7.7: The parameterization of the Search-Tree pattern for the Fifteen Puzzle in CO_2P_3S.

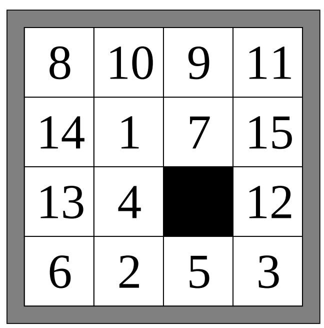


Figure 7.8: The puzzle solved using the Search-Tree pattern.

7.6.4 Results

A sequential application which searched the tree in a recursive manner was used to create the parallel version. The recursive method in the sequential program were reused by having the `conquer()` method "decorate" the method. In other words, the `conquer()` is a wrapper for a call to the sequential traversal method. Of the 125 lines of the original application, all of it was reused in the parallel version with only minor modifications necessary such as adding lines to use the generated framework. The sequential application consisted of 4 classes, all of which was reused. The parallel application contained 12 classes, 8 of which were generated and comprised 123 of the 308 line program. The pattern user wrote 44 lines of new code, most of which was restricted to the hook methods.

The application was used to solve the puzzle shown in Figure 7.8, selected from a standard set of 100 puzzles [15]. The machine used was an SGI Origin 2000 with 46 MIPS R100 195 MHz processors and 11.75 gigabytes of memory. A native threaded Java implementation from SGI (Java 1.3.1) was used with optimizations and JIT turned on, and the virtual machine was started with 1 GB of heap space. Trees nodes to a depth of 7 were expanded for the search tree.

Table 7.1 shows the results of using the Search-Tree pattern to find a solution to the puzzle. The times given are the averages of ten runs. The results show respectable speedups, though the performance begins to degrade after 4 processors. One factor which prevents better speedups is the overhead associated with the use of the `canContinue()` method. Recall that this method is placed in the `conquer()` method, so it is called for every iteration of the recursive sequential tree search. This is done so that processors are properly informed when another processor finds the goal and the calling processor can cease searching. A more critical factor is that a small number of `conquer()` nodes are responsible for the majority of the work. As more processors are used this load imbalance become more pronounced as demonstrated by the results.

Processors	1	2	4	8	16	32
Time (sec)	1578	905	443	235	149	115
Speedup	-	1.74	3.56	6.71	10.59	13.72

Table 7.1: Speedups and wall clock times for finding a solution to puzzle A. The times given are the averages of ten runs.

7.7 The Game of Kece

7.7.1 Description of the Problem

The game of Kece [4, 31] is a zero-sum game[1] similar to the game of Scrabble; two players alternately select words from a W-sized list and place the word onto a $N \times N$ board in a crossword-style fashion. The initial board state contains a single word for the players to begin, and words on the list are only shown to the players through a T-sized window (typically two or three). The game is finished when all words from the list are exhausted, or when no more words can be placed on the board. Each move consists of a player placing a word on the board such that it overlaps with a previously played word, and the player's score for a move is the number of words that are overlapped. Figure 7.9 shows a sample Kece board. Four words have already been placed on to the board, and the player has chosen the word "CONCURRENT" as the next word to place on the board. The word can be played in any of the indicated positions for scores of 1 or 2, as well as in other locations. As can be seen by the figure, scores of 1 are common, with scores of 2 or better being progressively harder to achieve. The other words in the window (the dotted box in Figure 7.9) could also have been played.

Zero-sum games are traditionally solved through game-tree search algorithms. The minimax search algorithm [23] simulates the moves of two players where for each of their moves they are trying to minimize their opponent's score and maximize their own. The algorithm searches a game tree in a depth-first fashion by assigning to a node the maximum value of the child nodes at even levels (assuming that the root node is at level 0), and the minimum value of the children to nodes at odd levels as shown in Figure 7.10. Eventually, the best score that the player can achieve propagates to the root node and the player can then follow that value down the branches of the tree.

However, for real games, *minimax search* is impractical as it has a time complexity of $O(b^m)$ for a tree of depth m and a branching factor of b. The alpha-beta search algorithm [23] improves this complexity by allowing the pruning of branches that can be shown to be inferior. Figure 7.11 shows an example of an *alpha-beta search tree*. Like minimax, alpha-beta search trees are searched in a depth-first fashion. The first pruning of a node occurs

[1] A zero-sum game is a game, such as Chess, in which for each move a player 'gains' by the same amount that the opponent 'loses'.

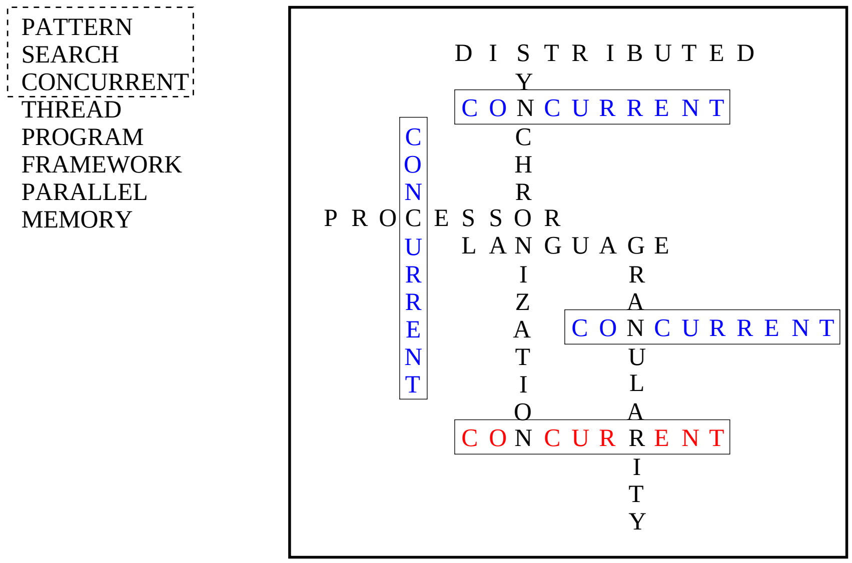


Figure 7.9: An example of a 20x20 Kece board where $N = 20$, $W = 12$, and $T = 3$. The initial word on the board was LANGUAGE and four words have already been placed on the board.

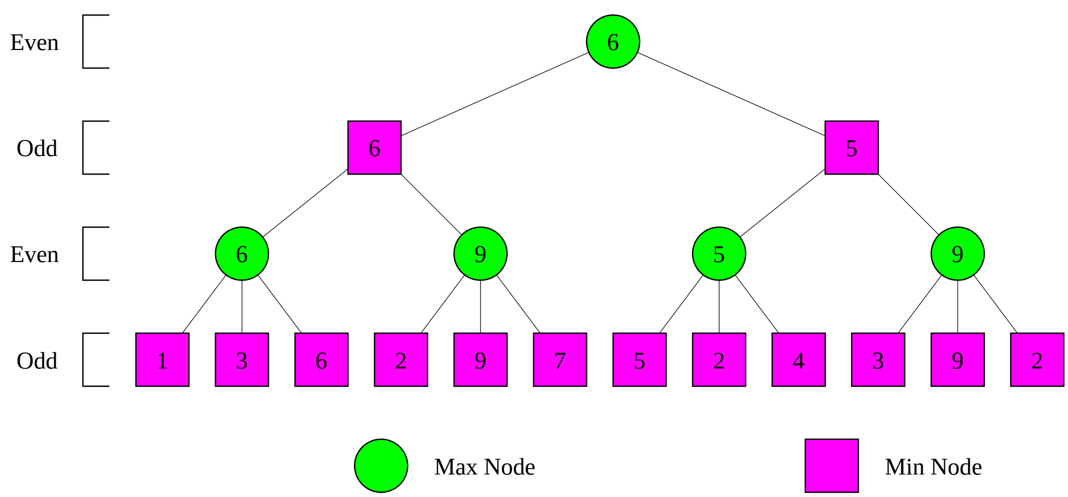


Figure 7.10: A minimax game tree.

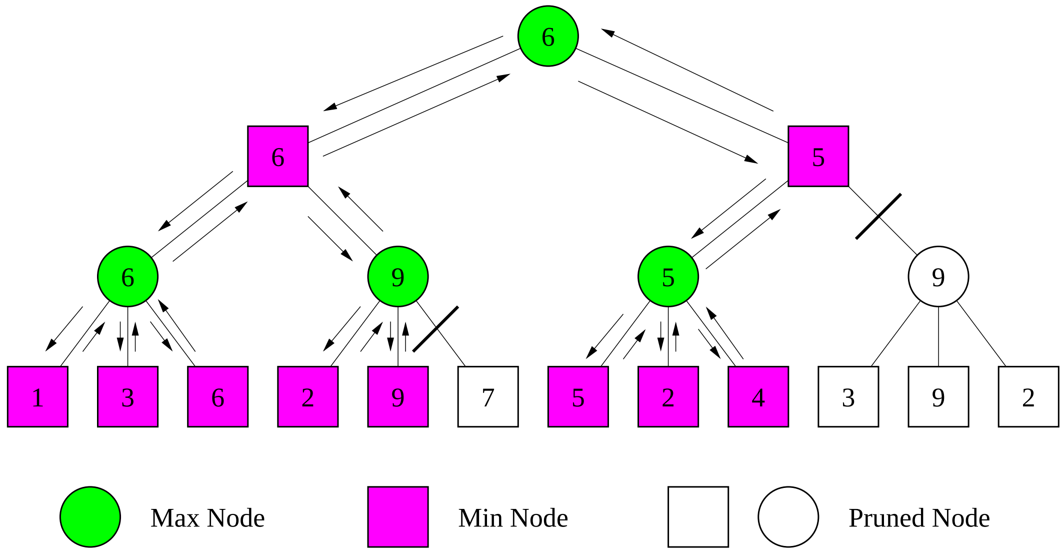


Figure 7.11: An alpha-beta game search tree. Recall that alpha-beta trees are searched depth-first from left to right.

due to the setting of the value 9 in the max node at level 2 on the left-hand side, which is greater than its min node parent of value 6. Therefore further searching of this subtree will not contribute to the solution as the node's value can only increase, and the remaining subtrees (in this case the node with value 7) can be pruned. Similarly on the right-hand side, the pruned subtree cannot contribute to the solution as the min node at level 1 can only become lower than 5 and will never replace the 6 at the root of the tree. Through the use of pruning the time complexity for searching the tree can be reduced to $O(b^{m/2})$.

Alpha-beta search appears to be easily parallelizable, as each branch of a subtree could be searched in parallel. However, this leads to a significant amount of work that would never have been done in the sequential case, due to less pruning [31]. One of the challenges of parallel alpha-beta search is for the processors to effectively communicate cutoff values to minimize the amount of unnecessary work. In actual fact, finding an efficient parallelization of alpha-beta search has been found to be a very hard problem [6].

7.7.2 Why the Search-Tree Pattern is Appropriate

As the Kece game is solved using an alpha-beta search, the Search-Tree pattern is the pattern in CO_2P_3S best suited for implementing it.

7.7.3 Using the Search-Tree Pattern

The parameterization of the Search-Tree pattern for the Kece problem is shown in Figure 7.12. As alpha-beta search is a depth first search, the *traversal* parameter is set to depth-first. Since we are looking for the optimal score, the entire tree must be searched (except branches which are pruned), so the *early-termination* parameter is set to false. The *done() verification* parameter was turned on during the testing of the program, but was turned off for the performance results given below.

For this problem the default implementation of `parallel()`, which expands nodes up to four levels deep in the tree, was sufficient. Larger depths can result in large work queues of nodes with small granularity. The `divide()` method creates new tree nodes for all the possible placements of the words in the word list window for a particular board. The

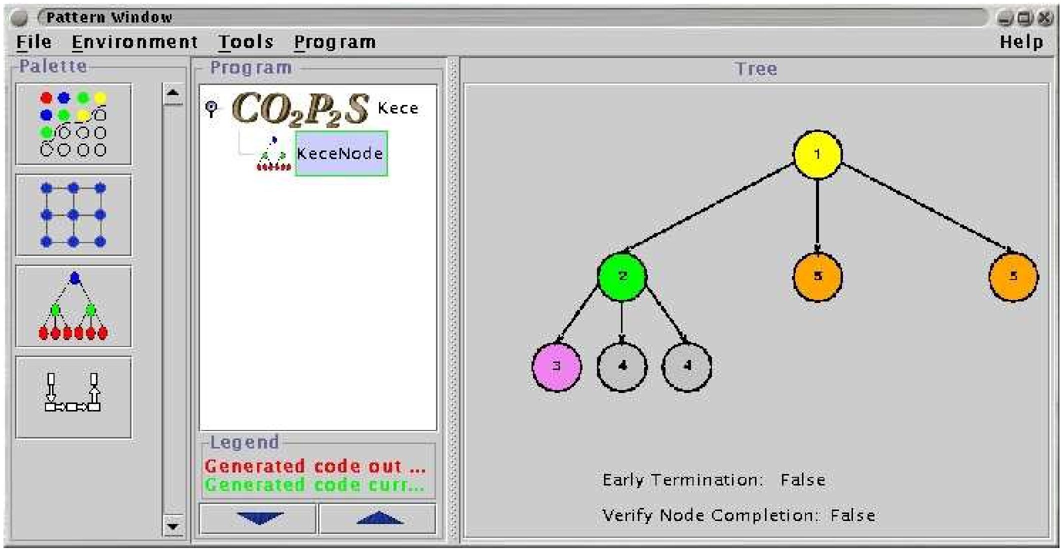


Figure 7.12: The Kece problem configuration of the Search-Tree pattern in CO_2P_3S.

`conquer()` method gathers alpha and beta values from the parent node to ensure that the most recent values are used, and proceeds with a recursive sequential traversal of the subtree. Depending on the type of the node the `updateState()` method gathers either the alpha or the beta value of the child and uses the value to set its own alpha or beta value. Finally, a node is considered `done()` if the number of times the node has been updated is equal to the number of children that node has, which is the default behaviour for this method.

7.7.4 Results

The creation of a parallel version of a sequential Kece application involved extending the sequential program by implementing the hook methods. Most of the 375 lines of code from the sequential application were reused with only minor revisions. Of the 16 classes in parallel application, 8 were generated and the rest were from the sequential application with the only modifications being to the mainline class in order to use the Search-Tree pattern. The final application was 539 lines in length with 135 lines being from the generated framework. Only 42 new lines of code were required to implement the parallel version from the sequential version.

The Kece program was run with $N = 20$, $W = 10$, and $T = 3$, with the board initially containing a single word. Nodes up to four levels deep were expanded and their children were searched in parallel. The hardware/software configuration used was the same as for the Fifteen Puzzle program (see Section 7.6). Table 7.2 shows the results of taking the median of 10 runs of the program.

As Table 7.2 shows, the application achieves good speedups up to 4 processors. After this point there is a marked decline in performance. This occurs due to the technique used for the depth-first search. Recall that in a depth-first search, no other nodes are processed until the left-hand side of the division tree is processed. Therefore, until the `conquer()` of the left-most child is finished, all other processors remain idle. Performance could therefore be improved if the granularity of the left-most node is reduced by expanding nodes to a deeper level of the tree. While this would minimize the amount of time that the other threads wait, it would also substantially increase the memory requirements of the application as the number of nodes in the queues would grow exponentially with respect to the depth.

Processors	1	2	4	8	16	32
Time (sec)	2159	1121	631	447	372	298
Speedup	-	1.93	3.42	4.83	5.80	7.24

Table 7.2: Speedups and wall clock times for a Kece game with $N = 20$, $W = 10$, and $T = 3$.

The depth at which nodes are no longer divided was extended from four to six in an attempt to reduce the amount of time that processors were left idle while the left-hand child was being processed. However, no improvement was shown. This is likely due to another limitation which is outlined next.

Another restriction to the performance of this application is the number of siblings for each left-hand child. Remember that once the left child is processed, all of its siblings are processed in parallel. If the number of siblings is less than the number of processors, then some processors still remain idle. Table 7.3 shows the number of siblings for the four levels of the search which were expanded.[2] Assuming that all nodes at a single level take the same amount of time to process, then for all these levels the number of siblings is such that many of the 16 or 32 processors will remain idle at each level. This is not as much of a problem at the lower levels (levels 3 and 4) where the granularity of the nodes is small as at the higher level (levels 1 and 2) where the granularity of nodes could be quite large. However, the previous assumption that all nodes require the same amount of work is not true. Due to pruning effects, nodes require disproportionate amounts of processing. This makes the problem even worse as it is even more likely that processors will remain idle. The fact that changing the granularity of the left-most child showed no effect supports that the number of siblings is the primary consideration in the poor speedups of this application.

Depth	root	1	2	3	4
Siblings	0	23	29	23	17

Table 7.3: Number of siblings at various tree depths for the Kece tree.

7.8 Summary

The Search-Tree is a new pattern to CO_2P_3S which was used to parallelize a single-agent search application and an alpha-beta search application. The first application was used IDA* for finding a solution to the Fifteen Puzzle. The second was an alpha-beta tree search for the game of Kece. Given sequential versions of these two applications, the creation of parallel versions using the pattern required writing less than 50 lines of new code for each application, as most of the sequential code was reused. While the parallel Kece application showed poor speedups for more than four processors, this was found to be a result of the shape of the game tree. While the parallel version of the fifteen puzzle application showed better speedups, the load imbalance of the `conquer()` nodes and the overhead of the `canContinue()` method limited the performance of the application.

[2]Further depths showed similar numbers of siblings.

Chapter 8

Contributions, Future Work, and Conclusions

8.1 Contributions

The primary contribution of this research is the addition of two new parallel patterns to the CO_2P_3S system. The first is the Wavefront pattern that is used for applications in which there exists data dependencies between elements of computation such that a computation of an element cannot proceed until its prerequisites have been satisfied. The second is the Search-Tree pattern which provides a framework for parallelizing breadth-first and depth-first search trees.

The addition of the Search-Tree pattern also introduces a new type of parameter: the validation parameter. This parameter type allows a pattern designer to include in a pattern framework code which can be generated to verify the proper use of the framework.

Another contribution of this research was the validation of MetaCO_2P_3S. The creation of the Wavefront and Search-Tree patterns provided the first opportunity to add new patterns to CO_2P_3S. Previously, the patterns which were already part of the the system had been re-implemented using the tool.

The final contribution of this work is the demonstration that CO_2P_3S has sufficient utility and extensibility to aid in the development of concurrent solutions for a range of problems. In doing so it was also shown that given the right pattern, developing a concurrent solution which scales reasonably, building from a sequential program, requires little effort.

8.2 Future Work

The patterns in CO_2P_3S are not immutable and may evolve as they are used for new applications and as new parameter types or values are found. Recall that in the evolution of the Wavefront pattern, each new application contributed to its evolution, but in diminishing degrees. Following are a few suggestions of possible improvements to the patterns highlighted in this work:

The Mesh Pattern. The Mesh pattern could be extended to allow mesh elements to be primitive types. The Wavefront pattern contains this feature as a performance parameter and the Mesh pattern may similarly benefit.

The Pipeline Pattern. Currently the user only specifies the names of the abstract classes for the ordered and unordered stages. A possible improvement is to allow the user to also specify the names and type of the stages.

The Wavefront Pattern. The dependency set for the Wavefront pattern is restricted to those which fall within a 90° arc. There may exist an application for which this is too restrictive, such as one having a dependency set of North, Southeast, and an arc of 180° may be necessary. However, this would make the pattern framework code much more complex as there would be 64 unique dependency sets which may be specified. However, an application must first be found that requires this functionality.

The Search-Tree Pattern. Currently the decision of when to cease dividing tree nodes is based on the depth of the node in the tree. A possible performance parameter would allow the user to select either the current tree-depth criteria, or a criteria based on the number of idle threads. This may produce an application which has better load-balancing, such as for the Fifteen Puzzle.

The image thinning problem is described as having two portions [10, 31]. The second portion of the problem, called *skeletonization*, involves identifying the connected components of the image. This was not done as it was felt that doing so would not contribute significantly to the focus of this work, which was demonstrating that CO_2P_3S could implement the problem. The parameterization of the Mesh pattern would be the same for the skeletonization portion as for the thinning portion. However, this application provides the opportunity to examine the composability of CO_2P_3S patterns. This was one of the characteristics for an ideal parallel programming system put forth by Singh *et al.* While CO_2P_3S patterns are designed to be easily composable, there has not been much work done to demonstrate the composability of these patterns. The only previous work in this direction was the composition of the Distributor and Phases patterns in a PSRS[1] application [18].

8.3 Conclusions

While parallel programs are known to improve the performance of computationally-intensive applications, they are also known to be challenging to write. Parallel programming tools, such as CO_2P_3S, provide a way to alleviate this difficulty. The CO_2P_3S system is a relatively new addition to a collection of such tools. Before it can gain wide user acceptance there needs to be confidence that the tool can provide the assistance necessary. To this end the utility of the CO_2P_3S system was tested by implementing a variety of problems. If the CO_2P_3S system did not provide the means to create a solution for a particular problem, then the extensibility of the tool was tested by attempting to add new patterns to CO_2P_3S.

The Cowichan Problems were chosen as a good generalization of the range of applications that a parallel programming system should support. Of the seven problems presented in this problem set, three were found to use patterns already existing in CO_2P_3S. These were the Turing Ring, image thinning, and map overlay problems. The remaining four applications resulted in the creation and addition of two new patterns. The Wavefront pattern was created to implement a skyline matrix solver and a matrix product chain solver. The Search-Tree pattern was added to CO_2P_3S in order to implement a solutions for the Kece game and the Fifteen puzzle.

Assuming that the Cowichan Problems provide a sufficiently wide ranging collection of the problems, then the CO_2P_3S system has the sufficient utility, extensibility, and usability to develop concurrent applications.

[1] Parallel Sorting by Regular Sampling.

Bibliography

[1] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steve Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. *Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2002. Awarded Best Paper of the workshop.

[2] Alberto Bartoli, Paolo Cosini, Gianluca Dini, and Cosimo Antonio Prete. Graphical design of distributed applications though reusable components. *IEEE Parallel and Distributed Technology*, 3(1):37–51, 1995.

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[4] Peter Boncz. Parallelizing the crossword generation game in Orca. Student project report, Vrije Universiteit Amsterdam, May 1994.

[5] David S. Bouman. Parallelizing a skyline matrix solver using Orca. Student project report, Vrije Universiteit Amsterdam, August 1995.

[6] Mark Brockington and Jonathan Schaeffer. APHID: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing*, 60:247–273, 2000.

[7] Steve Bromling. Meta-programming with parallel design patterns. Master's thesis, Department of Computing Science, University of Alberta, 2002.

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

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

[10] Rudolf S. de Boer. Parallel thinning and skeletonization using Orca. Student project report, Vrije Universiteit Amsterdam, August 1994.

[11] Geoffrey Dutton, editor. *Harvard Papers on Geographic Information Systems: Volume 6 — Spatial Algorithms: Efficency in Theory and Practice*. Laboratory for Computer Graphics and Spacial Analysis, 1978.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Dhrubajyoti Goswami, Ajit Singh, and Bruno R. Priess. Architectural skeletons: The re-usable 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.

[14] Dhrubajyoti Goswami, Ajit Singh, and Bruno R. 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.

[15] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[16] H. F. Langendoen. Parallelizing the polygon overlay problem using Orca. Student project report, Vrije Universiteit Amsterdam, August 1995.

[17] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, second edition, 2000.

[18] Steve MacDonald. *From Patterns to Frameworks to Parallel Programs*. PhD thesis, Department of Computing Science, University of Alberta, 2001.

[19] Steve MacDonald, Duane Szafron, and Jonathan Schaeffer. Object-oriented pattern-based parallel programming with automatically generated frameworks. *Proceedings of the 5th USENIX Conference on Object-Oriented Tools and Systems*, pages 29–43, May 1999.

[20] Steve MacDonald, Duane Szafron, Jonathan Schaeffer, and Steve Bromling. Generating parallel program frameworks from parallel design patterns. *Proceedings of the 6th International Euro-Par Conference*, pages 95–104, August 2000.

[21] Dion Nicolaas. Parallelizing the Turing ring using Orca. Student project report, Vrije Universiteit Amsterdam, August 1994.

[22] Jorg Nolte, Yutaka Ishikawa, and Mitsuhisa Sato. TACO - prototyping high-level object-oriented programming contructs by means of template based programming techniques. *ACM SIGPLAN Notices*, pages 35–49, December 2001.

[23] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 5. Prentice Hall, 1995.

[24] Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[25] Martin Schuetze, Jan Peter Riegel, and Gerhard Zimmermann. A pattern–based application generator for building simulation. In *Proceedings of the Sixth European Software Engineering Conference (ESEC'97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 468–482. Springer–Verlag, 1997.

[26] Ajit Singh, Jonathan Schaeffer, and Duane Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.

[27] Anil R. Sukul. Parallel implementation of an Active Chart parser in Orca. Student project report, Vrije Universiteit Amsterdam, August 1995.

[28] Ladan Tahvildari and Ajit Singh. Impact of using patten-based systems on the qualities of parallel applications. *Proceedings of IEEE International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, Vol. III:1713–1719, June 2000.

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

[30] Alan M. Turing. The chemical basis of morphogenesis. *Transactions of the Royal Society of London*, 237:37–42, 1952.

[31] Gregory V. 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, April 1994.

[32] Gregory V. Wilson and Henri E. Bal. An empirical assessment of the usability of Orca using the Cowichan problems. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.

[33] Gregory V. Wilson and R. Bruce Irvin. Assessing and comparing the usability of parallel programming systems. Technical Report CSRI-321, University of Toronto, 1995.