Available online at www.sciencedirect.com

SCIENCE DIRECT®

J. Parallel Distrib. Comput. 65 (2005) 1542 – 1557

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Asserting the utility of CO_2P_3S using the Cowichan Problem Set

John Anvik[a,*], Jonathan Schaeffer[b], Duane Szafron[b], Kai Tan[b]

[a]*Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada V6T 1Z4*
[b]*Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8*

Received 6 February 2004; received in revised form 21 March 2005; accepted 31 May 2005
Available online 9 August 2005

Abstract

Parallel programming environments provide a way for programmers to reap the benefits of parallelism, while reducing the effort required to create parallel applications. The CO_2P_3S parallel programming system is one such tool that uses a pattern-based approach to express concurrency. Using the Cowichan Problems, we demonstrate that CO_2P_3S contains a rich set of parallel patterns for implementing a wide variety of applications running on shared-memory or distributed-memory hardware. An example of these parallel patterns, the Search-Tree pattern, is described and it is shown how the pattern was used to solve the Fifteen Puzzle problem. Code metrics and performance results are presented for the Cowichan applications to show the usability of the CO_2P_3S system and its ability to reduce programming effort, while producing programs with reasonable performance.

Keywords: Parallel programming; Programming environments; Design patterns; Cowichan Problems; CO_2P_3S; Fifteen Puzzle; Search-Tree pattern

1. Introduction

In many fields of research there exist problems which simply take too long to solve using a single processor. Only by dividing the problem into separately computable components and using multiple processors can these problems be solved in a reasonable time frame.

However, doing so is not without cost. Adding parallelism also adds new concerns to the application, such as synchronization and communication between the processors. This leads to either an increased complexity of the algorithm, or the use of a completely different algorithm. It also makes the debugging of these programs more difficult as non-determinism is now introduced. The writing of parallel programs is known to be a complex and error-prone task, even for experts in the field.

The state of the art in parallel programming tools is represented by OpenMP for shared-memory programs and MPI for distributed-memory programming. These are low-level models in that the user must explicitly represent the parallelism in the code. The programmer is required to adapt or restructure their application to accommodate the concurrency. In the case of MPI, this can translate into hundreds or more additional lines of potentially error-prone code.

There is hope however. There exist strategies in sequential programming which may be used across many problems. These strategies are called *design patterns* [15] and they encapsulate the knowledge of solutions for a class of problems. To solve a problem using a design pattern, an appropriate pattern is chosen and adapted to the particular problem. By referring to a problem by the particular strategy that may be used to solve it, certain design decisions are implicitly communicated and a deeper understanding of the solution to the problem is conveyed.

Just as there are sequential design patterns, there exist *parallel design patterns* which capture the synchronization and communication structure for a particular parallel solution. The notion of these commonly occurring parallel structures has been well known for decades in such forms as skeletons [3,11,16], or templates [5,20,23]. Examples of common

[*] Corresponding author.
E-mail addresses: janvik@cs.ubc.ca (J. Anvik), jonathan@cs.ualberta.ca (J. Schaeffer), duane@cs.ualberta.ca (D. Szafron), cavalier@cs.ualberta.ca (K. Tan).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.05.029

parallel design patterns are the fork/join model, pipelines, meshes, and work piles.

Unfortunately since design patterns are design constructs, they give a description of the solution but not the solution itself. After choosing a design pattern, the programmer must still adapt and implement the pattern for the specific application. If design patterns are at one end of the design spectrum, then frameworks are at the other end. Whereas a design pattern is an abstract description of a 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. To use a framework, *hook methods* are implemented to contain application-specific code, such as the function of a button. These hook methods are then called in the application through the framework. Unlike design patterns that must be re-implemented each time that they are used, frameworks provide a maximum of code reuse. However this re-usability comes at the cost of less generality.

The effort required to adapt a design pattern for a particular application may range from being trivial to being substantial. Consequently, many tools have been created to minimize programmer effort in this regard. Tools such as Budinsky's web-based tool [10], Together ControlCenter [12], PSiGene [24], and ModelMaker [27] are all examples of efforts made in this direction. Although there have been many academic attempts to build pattern-based high-level parallel programming tools like those for sequential design patterns, few have gained acceptance by even a small user community. The idea of having a tool that can take a selected parallel structure and automatically generate correct structural code is quite appealing. Typically, the user would only fill in application-dependent sequential routines to complete the application. Unfortunately, these tools have not made their way into practice for a number of reasons:

(1) *Performance*: Generic patterns produce generic code that is inefficient and suffers from loss of performance.
(2) *Utility*: The set of patterns supported by a given tool is quite limited, and if the application does not match the provided patterns, then the tool is effectively useless. Furthermore, a tool may only be suitable for a single type of parallel architecture.
(3) *Extensibility*: High-level tools often contain a fixed set of patterns and the tool cannot be extended to include more.

The CO_2P_3S [1] parallel programming system uses a design patterns approach to ease the effort required to write parallel programs. The system addresses the limitations of previous high-level parallel programming tools in the following ways:

(1) *Performance*: CO_2P_3S uses *adaptive generative parallel design patterns*. An adaptive generative design pattern is an augmented design pattern which is parameterized so that it can be readily adapted for many applications. It is used to generate a parallel framework tailored for each specific application. In this manner the performance degradation of generic frameworks is eliminated.
(2) *Utility*: CO_2P_3S provides a rich set of parallel design patterns, including support for both shared and distributed memory environments.
(3) *Extensibility*: MetaCO_2P_3S is a tool used for rapidly creating and editing CO_2P_3S patterns. CO_2P_3S currently supports 15 parallel and sequential design patterns, with more patterns under development.

This paper focuses on the utility aspect of CO_2P_3S. The performance aspect of using CO_2P_3S has already been presented [9]. Our intent is to show that the use of a high-level pattern-based parallel programming tool is not only possible, but more importantly, it is practical. The CO_2P_3S system can be used to quickly generate code for a diverse set of applications with widely different parallel structures. This can be done with minimal effort, where effort is measured by the number of additional lines of code written by the programmer using CO_2P_3S. The Cowichan Problems [29] are used to demonstrate this utility by showing the breadth of applications which can be written using the tool. Furthermore, it is shown that a shared-memory application can be recompiled to run in a distributed memory environment with no changes to the user-supplied code.

Test suites for assessing system *performance*, 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, we know of only one non-trivial set: the Cowichan Problems [29]. The Cowichan Problems are 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 [30]. 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.

We begin by providing an overview of the CO_2P_3S system and then show how the system can be used to implement a parallel solution to a common problem. The SearchTree pattern is then described in detail as an example of a generative design pattern. We then demonstrate the utility of CO_2P_3S by showing how the patterns in CO_2P_3S were used to write solutions for the Cowichan Problem Set.

2. 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

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

Design Patterns (PDP) methodology [18]. This methodology uses a layered programming model to produce structurally correct parallel programs. Unlike similar tools, CO_2P_3S allows the programmer access to the generated code to support tuning. The first layer is the Patterns Layer by which CO_2P_3S generates parallel programs through the use of *pattern templates*. A pattern template is an intermediary form between a pattern and a framework, and represents a family of design solutions. Members of the solution family are selected based upon the values of the options for the particular pattern template. 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 selection of pattern options.

The remaining two layers of the CO_2P_3S system are the Intermediate Code Layer and the Native Code Layer. These layers grant the programmer the opportunity to tune the performance of their application at two levels of abstraction. The Intermediate Code Layer provides a high-level, explicitly parallel object-oriented programming model, allowing the programmer to work with abstractions of things such as barrier synchronization and parallel loops. The Native Code Layer exposes the implementation of these abstractions. An example of the use of this layer is doing low level performance tuning.

The pattern options in CO_2P_3S are divided into four types: lexical, design, performance, and verification options. *Lexical options* are various class and method names in the pattern framework which are provided by the programmer. *Design options* are pattern options which affect the overall parallel structure of the generated framework. *Performance options* introduce optimizations that may improve performance by changing the internal framework code (however these changes are not visible to the programmer). *Verification options* allow for the inclusion of pieces of code in the framework to ensure the proper use of the pattern and find errors in the programmer's code.

A framework generated by CO_2P_3S provides the communication and synchronization for the parallel application, and the user provides the application-specific sequential code. These code portions are added through the use of sequential hook methods in the framework code. At this pattern layer, the tool abstracts away the parallelism from the application-specific portions and prevents the programmer from changing the code which implements the parallelism, thereby maintaining the correctness of the parallel code. Recall however that through CO_2P_3S's layered model the user has access to lower abstraction layers when necessary in order to tune the automatically generated code.

Extensibility of a programming system improves its utility. CO_2P_3S improves its utility by allowing new pattern templates to be added to the system using the MetaCO_2P_3S tool [8]. Pattern templates added through MetaCO_2P_3S are indistinguishable in form and function from those already contained in CO_2P_3S. 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 MetaCO_2P_3S supports its rapid addition to CO_2P_3S.

The descriptions of pattern templates generated by MetaCO_2P_3S are stored in system-independent XML format. This ensures that the patterns generated by MetaCO_2P_3S can be used not only by the CO_2P_3S system itself, but also by any template-based programming tool which uses XML. The creation of a system-independent pattern repository creates a synergy that can enhance the utility of all systems that use this format since more patterns can be developed and distributed.

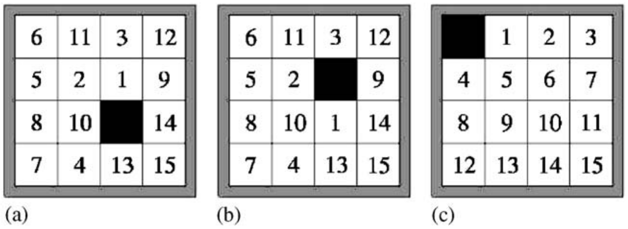
3. Using CO_2P_3S to implement IDA* search

To illustrate the CO_2P_3S system, we provide an example of how to generate a parallel program from a sequential program. Specifically, the IDA* search used in the Fifteen Puzzle [22] is implemented using the Search-Tree pattern.

3.1. The Fifteen Puzzle

Of the sliding tile puzzles, the Fifteen Puzzle is the one most frequently studied by artificial intelligence researchers. The puzzle consists of a 4 × 4 board with an arrangement of 15 numbered tiles and one blank space as shown in Fig. 1(a). The object of the game is to slide tiles into the blank space (Fig. 1(b)) in order to place all the tiles into numerical order (Fig. 1(c)), and to do so in the least number of moves.

To solve this problem, iterative deepening A* (IDA*) search is used [22]. The nodes of the search tree represent states (e.g. a game board configuration) and the arcs represent movement between states (e.g. a player's move). The cost of a node $f(n)$ is based on the heuristic value of a node given by $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path to a specific node from the root node and $h(n)$ is an estimate of how much farther it is to the goal. For this problem we count the number of moves needed to reach the state shown in Fig. 1(c). As long as $h(n)$ is *admissible* (i.e. never overestimates the distance to the goal), then IDA* is guaranteed to find a minimum cost path to the solution.



(a)

6	11	3	12
5	2	1	9
8	10	■	14
7	4	13	15

(b)

6	11	3	12
5	2	■	9
8	10	1	14
7	4	13	15

(c)

■	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig. 1. The Fifteen Puzzle: (a) an unsolved Fifteen Puzzle; (b) moving a tile in the Fifteen Puzzle; (c) a solved Fifteen Puzzle.

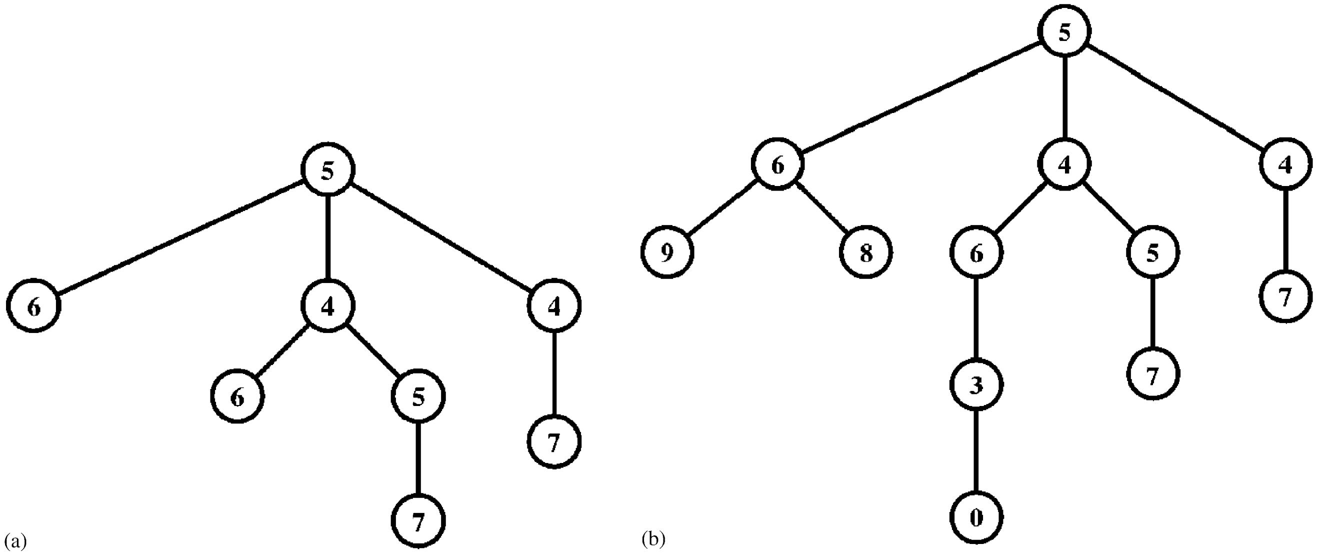


Fig. 2. An example of IDA*. The values shown in the nodes are the computed costs of the nodes (i.e. $f(n)$). (a) The initial search threshold is $h(n)$ where n is the root of the tree, 5 in this case. No solution is found for this threshold. (b) The threshold is incremented to 6 and the tree is searched again using the new threshold. A solution is found.

IDA* works by performing a depth-first search on iteratively larger trees. The search is controlled by a threshold value t, which is initially set to the $h(n)$ for the root node. The tree is then searched until either the goal is found or $f(n) > t$ 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. Fig. 2 shows an example of using IDA* to find a solution.

For the Fifteen Puzzle, $g(n)$ is the depth of a node in the tree and the sum of Manhattan distances is $h(n)$. 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.

3.2. *Solving the problem using* CO$_2$P$_3$S

To use the CO$_2$P$_3$S tool to generate parallel code for an application, the programmer first chooses an appropriate pattern. How a programmer determines the correct pattern for an application is a difficult problem and the subject of other research [19]. We assume that the user will select an appropriate pattern for their application. However, one of the benefits of using the CO$_2$P$_3$S tool is that if the user chooses the wrong pattern, they can choose another pattern and quickly generate a new version with little effort. As the Fifteen Puzzle uses IDA* search to find solutions, the Search-Tree pattern is chosen.

Once the programmer chooses the pattern, they then select the pattern options that best suit their application. The various options of the Search-Tree pattern are described in detail in Section 4.1. In this section we will just describe what options are selected for a solution to the Fifteen Puzzle.

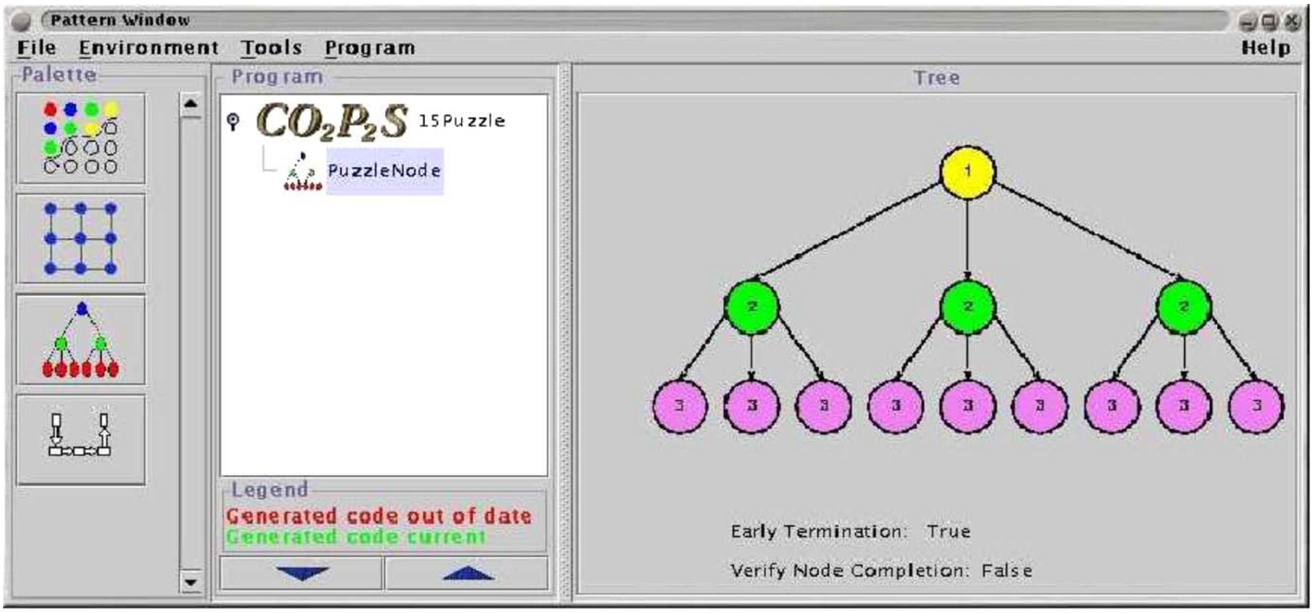
The first option to be provided is the lexical option for the class that represents a node of the search tree. We choose the name `PuzzleNode`. Next, the traversal technique to be used in searching the tree is chosen. As we would like to perform an IDA* search in parallel on multiple branches of the tree, we set the *traversal* option to `breadth-first`. Note that from the CO$_2$P$_3$S perspective this is a breadth-first search, but that the subtrees which are searched sequentially each perform depth-first searches.

Next, the *early termination* option is set to `true`. As soon as a solution is found the search should terminate.

The final option to be selected is the verification option. As explained in Section 4.1 the verification option verifies that the user's implementation of the `done()` hook method behaves correctly. Initially we set this option to `true`. Once we are assured that our `done()` method behaves appropriately, this option is set to `false` so that the verification code is removed from the generated framework for better run-time performance. Fig. 3 shows the parameterization of the Search-Tree pattern for the Fifteen Puzzle application.

Once all the options have been specified, we then generate the framework. Note that CO$_2$P$_3$S always generates a correct and fully functional framework. The hook methods of the generated framework are generated with default implementations so that the framework can be run “out of the box”, though it will typically not do anything useful at this point.

Once the framework has been generated, the programmer proceeds to fill in the hook methods with the application-specific code. The Search-Tree pattern generates five hook methods: `divideOrConquer()`, `divide()`, `conquer()`, `updateState(TreeNode child)`, and `done()`. These hook methods are described in Section 4.2.



Early Termination: True

Verify Node Completion: False

Fig. 3. The parameterization of the Fifteen Puzzle in CO$_2$P$_3$S. The left-hand pane shows the four patterns (Wavefront, Mesh, SearchTree, and Pipeline) currently loaded into the tool. The middle pane shows that the SearchTree pattern has been selected for the "15Puzzle" application and that the lexical option has been set to "PuzzleNode". The right-hand panel shows the selection of a breadth-first traversal, that early termination of the search is possible, and that no verification code is to be generated.

For this application the default implementation of `divideOrConquer()` is used. The `divide()` method creates new nodes for each possible move from a given position. The `conquer()` method is a wrapper method for the recursive traversal method from the sequential application. As the *early termination* parameter is set, two additional framework methods are provided to the programmer: `canContinue()` and `terminateAll()`. A call to `canContinue()` is added to the sequential method so that the processing of the node will stop if another node indicates that the goal was found by calling the `terminateAll()` method.

A node is considered "done" when it has received updates from all of its children. For this application, a counter is kept of the number of messages received and the `done()` method returns `true` when the counter equals the number of children. Finally, the `updateState()` method collects the nodes in the upper portion of the tree which are on the path toward the goal state.

In this application, nearly all of the 125 lines from the sequential application were reused. This is primarily due to the wrapping of the sequential traversal method in the `conquer()` hook method. As only a minor change was needed in the driver program, that too was almost entirely reused. Only 47 new lines of code were necessary to convert the sequential program into a parallel program, the majority of which were from implementing the hook methods.

4. The Search-Tree pattern

Four of the Cowichan Problems needed patterns that were not in CO$_2$P$_3$S when we began to implement them using CO$_2$P$_3$S. In any other pattern-based programming tool we would have had to stop at this point. However, as CO$_2$P$_3$S is an extensible programming system we were able to add the necessary patterns. Two new patterns were added to CO$_2$P$_3$S to solve the Cowichan Problems: the Wavefront Pattern and the Search-Tree pattern. The Wavefront pattern has already been described [1]. In this section a detailed description of the Search-Tree pattern is provided. Using the MetaCO$_2$P$_3$S tool, the CO$_2$P$_3$S system was extended to support this new pattern. Once the pattern had been designed, adding it to CO$_2$P$_3$S took approximately 9 h. The rapid addition of a new pattern to CO$_2$P$_3$S demonstrates how the extensibility of the system contributes to its utility.

4.1. Pattern options

The single lexical option for the Search-Tree pattern is the name of the class that will contain the hook methods which must be implemented by the pattern user.

This pattern has a single design option, 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 concurrently searched sequentially. This was the traversal technique used for searching the IDA* tree for the Fifteen Puzzle. If the tree is searched depth-first then all nodes on the left hand 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 concurrently processed. This order supports alpha–beta search [22] since

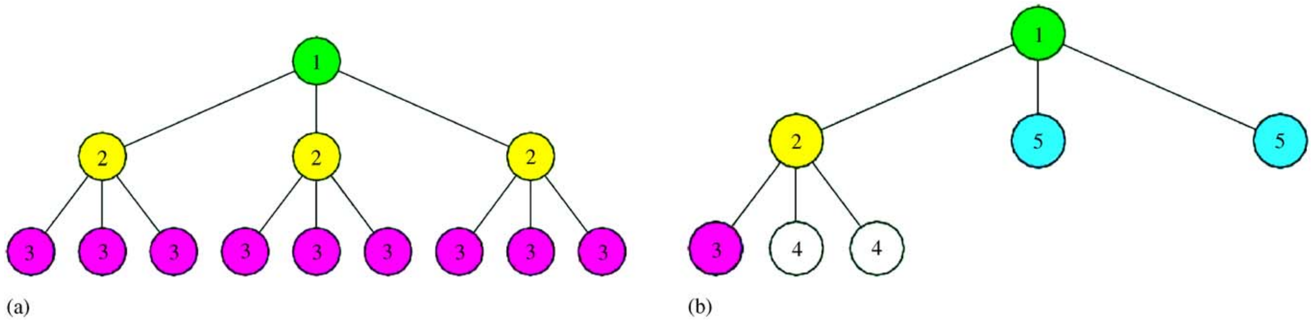


Fig. 4. Tree traversals in the Search-Tree pattern. Nodes with the same value are processed in parallel: (a) breadth-first traversal; (b) depth-first traversal.

the value of the node for the left child can be used to eliminate the search on some of the sibling nodes. Fig. 4 shows the order in which nodes are processed for both breadth- and depth-first parallel searches. Another possible traversal is best-first. However we add options and option values to CO$_2$P$_3$S patterns on a need-only basis and we do not yet have an application that requires a best-first traversal. We take this approach to prevent the generation of an overly general framework or the unnecessary explosion of option combinations.

The Search-Tree pattern has one performance option, *early termination*. This option allows for the termination of the search to occur before all nodes have been searched, such as when an application wants to terminate after finding one solution as opposed to all solutions.

The Search-Tree pattern introduced a new option type to the CO$_2$P$_3$S system called a *verification* option. In the Search-Tree pattern, the verification option verifies that the user's `done()` method behaves correctly. 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 is still waiting for the completion of its children, but the framework can detect that all the children have finished, then this indicates a fault in the user's code and an exception is thrown. This does not prevent the user from specifying the `done()` method to allow a sub-tree traversal to halt before all of the node's children have been processed. It simply prevents the application from waiting for more child nodes to be processed when they have all been processed.

4.2. Pattern hook methods

The CO$_2$P$_3$S parallel-pattern framework generated for the Search-Tree pattern contains five hook methods. These are the methods into which the programmer inserts the sequential code. Depending on the option settings, two additional framework methods may be generated which the user can make use of in their code, as explained and demonstrated in Section 3.2. Only a description of these methods is given here; how the hook methods are used in the Search-Tree framework is deferred until Section 4.3. The generated hook methods are:

`divideOrConquer()` This method indicates whether to generate a node's children (i.e. call `divide()`), who will then be processed in parallel or to proceed with the sequential computation of the node (i.e. call `conquer()`).

`divide()` This method generates a node's children.

`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 a child has completed its computation, it sends this message to its parent with itself as an argument.

`done()` This hook method specifies when a node is considered to be finished, such as when all children have updated their parent or when a child node finds a solution.

4.3. Implementation of the Search-Tree pattern

The default implementation of the `divideOrConquer()` hook method returns `true` if the depth of the node is less than a specified value to indicate that the node should be "divided." Otherwise, `false` is returned to indicate that the node should be processed sequentially. A default threshold for when to "divide or conquer" is used by the framework in the absence of the programmer providing a threshold via the framework constructor. The programmer may also override the default `divideOrConquer()` implementation if the decision to "divide or conquer" is based on something other than the depth of the node in the tree.

The Search-Tree pattern uses a work queue model 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. Computation of a node is accomplished via the `process()` method shown in Fig. 5. In the case of a 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 returned list is the left child and is placed immediately into the work queue. The remaining children are marked to indicate that they are dependent on that left child node

```
void process(TreeNode node)
{
    if(node is invalid) return

    if(divideOrConquer())
    {
        children = divide()

        // This code will only appear if the
        // breadth-first option setting is selected.
        if(breadth-first traversal)
            foreach child
                add child to work queue

        // This code will only appear if the
        // depth-first option setting is selected.
        if(depth-first traversal)
            mark first child as left child
            add left child to work queue
            foreach remaining child
                add to pending queue
    }
    else
    {
        conquer()
        update parent
    }
}
```

Fig. 5. Pseudo-code of the `process()` method.

and are placed onto the pending queue. When a node 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 condition and there were still nodes in the work queue to be processed. Processing of an invalid node returns immediately as shown in Fig. 5.

Note that while there are many tools which could produce code like that shown in Fig. 5, CO_2P_3S uses generative design patterns so that only the portions of the code relevant to the selected traversal method and other option settings would be generated. Once a traversal method has been selected, the other portions would not be generated, including the test for traversal type. This is a simple example of how generative design patterns can improve the performance of framework code via custom code generation.

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. Fig. 6 shows how updates are propagated up the tree and node completion is verified.

```
void update(Tree tree, TreeNode node)
{
    update state
    remove child from validation list

    // This code will only appear if the
    // depth-first option setting is selected.
    if(depth-first traversal)
        add nodes that are now ready to the
            work queue from the pending queue

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

Fig. 6. Pseudo-code of the `update()` method.

The user of the Search-Tree pattern is never aware of the above details. They are all internal to the generated framework, and the only view that the pattern user has is that of the application-specific hook methods. From the pattern user's perspective they select a set of option values, have CO_2P_3S generate the customized parallel framework code, and implement the necessary hook methods, as was shown in Section 3.2.

5. Using CO_2P_3S to implement the Cowichan Problems

The Cowichan Problems are 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- 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.

Table 1
Patterns used to solve the Cowichan Problems

Algorithm	Application	Pattern
IDA* search	Fifteen Puzzle	Search-Tree
Alpha–Beta search	Kece	Search-Tree
LU-Decomposition	Skyline Matrix Solver	Wavefront
Dynamic Programming	Matrix Product Chain	Wavefront
Polygon Intersection	Map Overlay	Pipeline
Image Thinning	Graphics	Mesh
Gauss-Seidel/Jacobi	Reaction/Diffusion	Mesh

Table 2
Code metrics (in lines of code) for the shared-memory implementations of solutions to the Cowichan Problems

Application	Seq.	Par.	Generated	Reused	New
Fifteen Puzzle	125	308	139	122	47
Kece	375	539	135	362	42
Skyline Matrix Solver	196	390	224	144	22
Matrix Product Chain	68	296	223	60	13
Map Overlay	85	455	235	60	160
Image Thinning	221	529	350	170	9
Reaction/Diffusion	263	434	205	177	52

Table 3
Speedups from 2 to 16 processors for the shared-memory implementations of solutions to the Cowichan Problems

Application	Problem size	2	4	8	16
Fifteen Puzzle	100 puzzles	1.74	3.56	6.70	10.60
Kece	$N = 20$, $T = 10$, $W = 3$	1.93	3.42	4.83	5.80
Skyline Matrix Solver	2000 × 2000, 50% full	1.93	3.89	7.84	14.86
Matrix Product Chain	2000 matrices	1.81	3.64	7.80	13.37
Map Overlay	3000 Hor./Vert. stripes	1.56	3.11	4.67	—
Image Thinning	3000 × 3000 pixels	1.88	3.53	6.39	10.43
Reaction/Diffusion	1680 × 1680	1.75	3.13	4.92	6.50

Table 4
Code metrics (in lines of code) for the distributed-memory implementations of solutions to the Cowichan Problems

Application	Seq.	Par.	Generated	Reused	New
Skyline Matrix Solver	196	1929	1760	144	25
Matrix Product Chain	68	1534	1458	60	16
Image Thinning	221	2150	1968	170	12
Reaction/Diffusion	263	1536	1304	177	55

For our work, one modification was made to the original problem set. The Cowichan Problems contain a single-agent search problem, the Active Chart Parsing Problem, that 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. When the problem was explored in 1995 using Orca [4], the sequential C version ran in 5.4 s for a sentence of 29 words [25]. This problem was appropriate when the Cowichan Problem Set was created, but modern processors have made it too easy. Therefore, a different single-agent search problem (IDA*), which was more representative of this class of problems, was selected.

All of the Cowichan Problems have been implemented using CO_2P_3S. Table 1 provides a summary of which CO_2P_3S pattern was used to solve each of the Cowichan Problems. Presented here is an overview of each of the patterns used to solve the problems and their options. For each problem a description of the problem and the pattern options selected for the solutions is provided. Tables 2 and 4, respectively, provide code metrics for shared- and distributed-memory solutions that were created using CO_2P_3S. Tables 3 and 5, respectively, show performance results for the solutions.

Table 5
Speedups for the distributed-memory implementation of the Reaction/Diffusion problem

Matrix size	Pattern layer	Native code layer
400 × 400	0.3	0.5
800 × 800	1.1	1.7
1200 × 1200	2.0	2.9

5.1. The Search-Tree pattern

As was demonstrated in Section 3, the Search-Tree pattern is a pattern used to parallelize tree search algorithms, such as those used in optimization and heuristic search, on shared-memory multiprocessors. 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

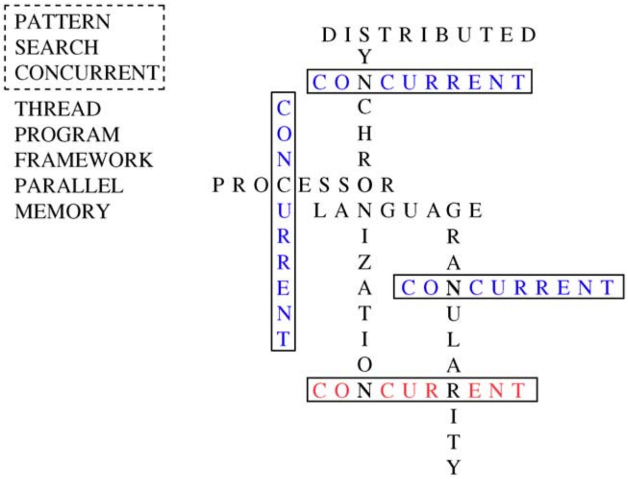


Fig. 7. An example of a 20 x 20 Kece board where $N = 20$, $W = 12$, and $T = 3$. The initial word on the board was LANGUAGE and four words have already been placed on the board. Four possible placements of CONCURRENT are shown; three that score one point and one that scores two points.

processor (conquer). A detailed description of this pattern was presented in Section 4.

5.1.1. The game of Kece

The game of Kece [6,29] is a zero-sum game[2] 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 (see Fig. 7). The initial board state contains a single word for the players to begin with 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.

Our solution for Kece uses alpha–beta search [22] to search the game tree of possible orders of word placements. Since alpha–beta search is a depth-first search, the *traversal* parameter is set to depth-first. As we are looking for the maximal score, the entire tree must be searched (except for branches pruned due to the alpha–beta search), so the *early-termination* parameter is set to `false`. As with the Fifteen Puzzle solution, the *done() verification* parameter was turned on during the testing of the program, but was turned off for the final version.

5.1.2. The Fifteen Puzzle

The Fifteen Puzzle and how CO_2P_3S was used to implement a parallel solution has already been described in Section 3.

5.2. The Wavefront pattern

The Wavefront pattern [1,2,26] 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 has been satisfied. While a wavefront may occur in arbitrary DAGs, the Wavefront pattern restricts the set of dependency graphs to those which occur in a matrix, as this appears to be the common case. Parallelism in the Wavefront pattern results from elements on the wavefront being data independent of each other, otherwise the elements could not be executed in parallel. CO_2P_3S contains versions of the Wavefront pattern for both shared-memory [2] and distributed-memory architectures [26].

The Wavefront pattern contains a single lexical option: 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 options. The design options for the pattern are:

(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 Fig. 8(a)),
Triangular where only the elements in the top triangular portion are computed (see Fig. 8(b)), and
Banded where the values that are computed are centered around the diagonal (see Fig. 8(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) of it. 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

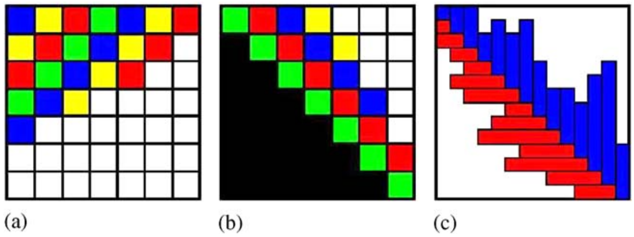
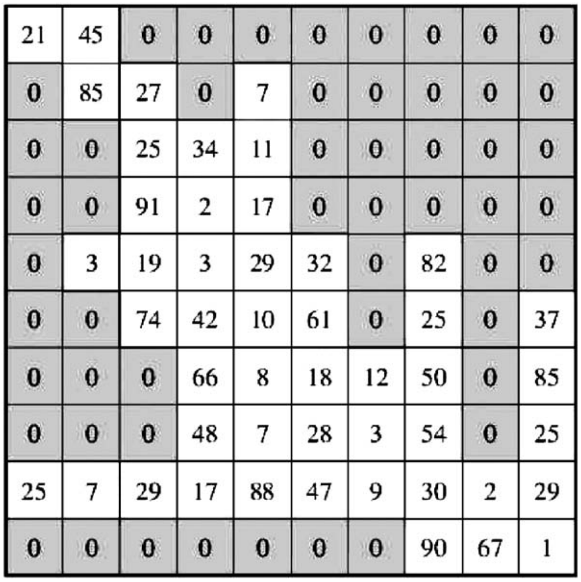


Fig. 8. Matrix shapes in the Wavefront pattern: (a) A full matrix shape; (b) a triangular matrix shape; (c) a banded matrix shape.

[2] 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'.



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

Fig. 9. Example of a 10 × 10 skyline matrix.

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 has the following performance options:

(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 option.

5.2.1. A skyline matrix solver

A skyline matrix is an $N \times N$ matrix in which each row has an indentation up to some distance from the diagonal and each column has a similar indentation. More formally, there exists constants r_i and c_i such that for $1 \leqslant r_i \leqslant i$ and $1 \leqslant c_j \leqslant j$ each row i has non-zero values from r_i to i, and each column j has non-zero values from c_j to j [29]. While any matrix may be viewed as a skyline matrix, 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. Fig. 9 shows an example of a skyline matrix.

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 [7]. The value of a matrix element a_{ij} is found by either Formula 1 of Fig. 10 if it occurs in the upper triangular portion of the matrix or by Formula 2 if it occurs in the lower triangular portion.

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

$$l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{ij})/u_{jj} \quad \text{for } 1 \leq i \leq n, 1 \leq j \leq i \qquad (2)$$

Fig. 10. Formulas for Doolittle's method of LU-decomposition.

The CO_2P_3S option 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. The performance options are set to `double` element type and Push notification.

5.2.2. Solving matrix product chains

Given a program that must multiply a series of M_i matrices for $0 \leqslant i \leqslant N$ and each matrix has r_i rows and c_i columns, in what order must the matrix multiplications be done so that the minimum number of scaler multiplications[3] is performed? For example, if the matrices A, B, C, D have dimensions 30 × 17, 17 × 12, 12 × 23, and 23 × 16, the number of scalar multiplications can range from 25,440 for $((AB)C)D$ to 15,840 for $A(B(CD))$.

Finding the optimal sequence is accomplished 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 [29]. The minimum cost is found by determining the least cost for finding the products $M_i \dots M_k$ and $M_{k+1} \dots M_j$ and using these to find the cost of the product for $M_{i \to k}$ and $M_{k+1 \to j}$. Once the upper portion is filled, the minimum cost will be the value in the top right-hand corner.

For the Matrix Product Chain the dependency set is {S, W}, the matrix type is Triangular, and neighbours-only is false. The performance parameters are set to Push notification and `int` element type.

5.3. The Mesh pattern

The Mesh pattern [18] is used for computing elements of a regular, rectangular two-dimensional data set where each element is dependent on its surrounding values and changes over time. 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

[3] The formula $r_i c_i c_{i+1}$ gives the number of scalar multiplications performed to find the product of two matrices.

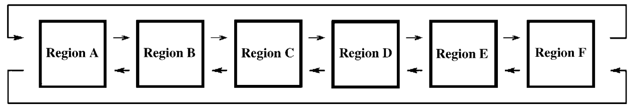


Fig. 11. Example of a Turing Ring system. The arrows indicate the flow of data between regions.

$$dX_i/dt = X_i(\rho_X + \alpha_X X_i + \beta_X Y_i) + \mu_X(X_{i+1} + X_{i-1} - 2X_i) \quad (3)$$

$$dY_i/dt = Y_i(\rho_Y + \alpha_Y X_i + \beta_Y Y_i) + \mu_Y(Y_{i+1} + Y_{i-1} - 2Y_i) \quad (4)$$

Fig. 12. Differential equations used in the Turing Ring model.

from either the cardinal points (north, south, east, and west) or all eight directions, and each element must be recomputed many times. This class of application includes programs for weather prediction and particle simulation.

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. Boundary values are then exchanged between partitions and another iteration is done. This continues until a local stopping condition is satisfied for all elements.

As with the Wavefront pattern, CO_2P_3S contains both shared-memory [18] and distributed memory implementations of the Mesh pattern [26].

The Mesh pattern has two lexical options: the class name for the mesh object and the class name for the elements of the mesh. The two design options for this pattern are the topology of the mesh and the number of neighbouring elements accessed to compute an individual element. The programmer may choose either a fully toroidal, a non-toroidal, a vertically toroidal, or a horizontally toroidal topology for the mesh. A given element may use results either from the cardinal points or from all eight directions. The performance option controls the amount of synchronization between iterations across the mesh. If an ordered mesh is selected, then an element waits for all other elements in the mesh to finish their respective computations before computing its next iteration. In a chaotic mesh, the elements compute their value as soon as possible.

5.3.1. The reaction/diffusion 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 [28].[4] The problem was later generalized to include other reaction-migration problems, such as those found in ecology, epidemiology, and petroleum engineering [21]. A model of such a system is shown in Fig. 11. Fig. 12 shows the pair of coupled differential equations which model the populations X_i and Y_i for each cell in the ring.

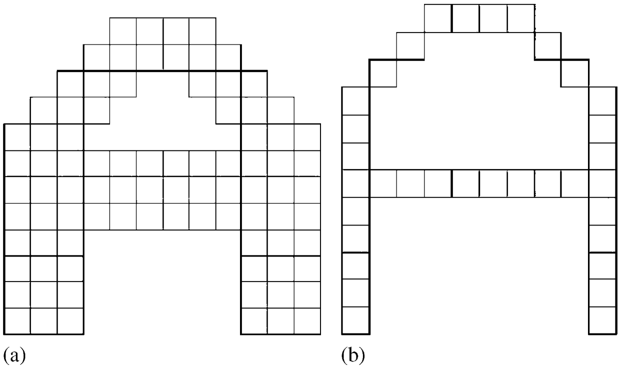


Fig. 13. An example of image thinning. The squares represent pixels: (a) an image before thinning; (b) the image after thinning.

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 coefficients, the model can be used to represent the mixing of water and oil in porous rock, the reaction/diffusion of two chemicals, or the progression of a disease through a population.

The reaction–diffusion solution that was produced used a fully toroidal mesh with elements acquiring data from their neighbours to the north, south, east, and west for computation. As the problem specifies that Jacobi iteration is to be used, the performance parameter was set to ordered.

5.3.2. Image thinning

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 [13,29]. Fig. 13 shows an example of this process.

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 line segment.
(2) Deleting the pixel 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 a mask fits then the pixel may be removed.

As the thinning operation of an image does not wrap around to the other side of the image, a non-toroidal mesh was used. Applying the pixel masks requires information about all pixels surrounding a given pixel, so the 8-point

[4] The Cowichan Problems calls this “The Turing Ring Problem”.

mesh option was selected. Finally, a proper thinning requires an ordered computation, so the performance parameter was set to ordered.

5.4. The Pipeline pattern

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. Abstractly, a pipeline can be regarded 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 of the following stages. Each stage of the pipeline can be viewed as having an object in a certain state, and the transition between pipeline stages is simply a change of state for the object [18].

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 [18] in CO_2P_3S is a pattern for shared-memory multiprocessors that 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, as stated previously, 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 manner the load is balanced across the pipeline.

The Pipeline pattern is unlike other CO_2P_3S patterns as it is a structural pattern. In other words, the pattern does not contain any design, performance, or verification parameters, but only lexical parameters. Whereas the other patterns produce a framework which is tuned for a particular application, this pattern generates the structural framework of a pipeline for the user. The user then specifies the pipeline stages by subclassing generated framework classes.

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 a first-in-first-out (FIFO) ordering of items. An unordered stage removes this restriction and allows items to be processed out of order. The use of unordered buffers is an optimization of the pipeline, and a user should only place an ordering on the work items when it is necessary for correctness.

5.4.1. Generating Map Overlays

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 is a problem that regularly occurs in spatial information systems [14,17,29]. Imagine that map A represents types of trees and map B represents types of soil. By combining these two maps, a new map is created which indicates the type of soil in which particular trees grow. Fig. 14 shows an example of overlaying two maps.

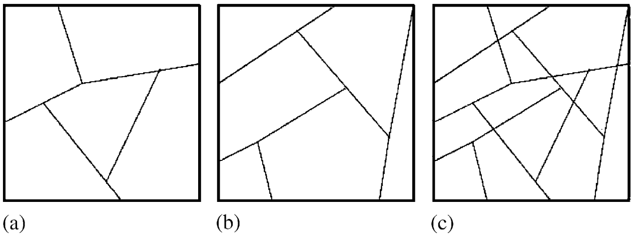


Fig. 14. An example of the Map Overlay problem: (a) map of tree type; (b) map of soil type; (c) map of tree and soil type.

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).

5.5. The effects of using CO_2P_3S

The results of using CO_2P_3S to implement solutions to the Cowichan Problems are presented here. These results take on two forms: code metrics to show the effort required by a user to take a sequential program and convert it into a parallel program, and performance results. Together, these results show that with minimal effort on the part of the user, reasonable speedups can be achieved. The speedups are not necessarily the best that can be achieved, since the applications could be further tuned to improve performance using the CO_2P_3S layered model [18]. Tables 2 and 3 show the results of using CO_2P_3S to generate shared-memory frameworks, and Tables 4 and 5 show the results for generating distributed-memory frameworks.

5.5.1. Shared-memory solutions

Recall from Section 3 that for the programmer to transform a sequential solution for the Fifteen Puzzle into a parallel solution using CO_2P_3S meant that the programmer had to write very little code. Table 2 shows this is not a unique case. It demonstrates that for a variety of problems a sequential Java program can be adapted to a shared-memory version with little additional effort on the part of the programmer. The time required to move from a sequential implementation to a parallel implementation took in the range of a few hours to a few days in each case.

While the sequential C code used in the Orca study [30] was available, the sequential solutions were completely rewritten in Java in order to use CO_2P_3S and to gain a better understanding of the problems. They were written by hand based on the descriptions of the problems [29] before CO_2P_3S was used to create the parallel implementations. This was done to try and minimize their tailoring to the CO_2P_3S system. However, as we knew that the programs would be used with CO_2P_3S we cannot be certain that all biases were removed.

The first two columns of Table 2 show the size of the sequential and parallel programs. Note that the parallel versions are 1.5 to 5 times larger than their sequential counterparts. This is due to the generation of all the necessary code for the spawning, synchronization, and load balancing of the processes that would normally be written by the programmer. The amount of code generated by CO_2P_3S is shown in the third column. If this same code had been written by the programmer, they would likely have to debug all this code before they had a working program. This would require the programmer to include error checking code in addition to what was required for the parallelization. As the parallel code is generated by the CO_2P_3S tool, this debugging has already been performed and the programmer has a working parallel program that is free of extraneous error checking.

The next column of Table 2 shows how much code was reused without modification from the sequential program. This reuse was either the result of directly reusing classes created for the sequential application or from cutting and pasting code into the hook methods.

The last column of Table 2 shows how much new code was required by the programmer to achieve a working solution to the problems. The additional code that the user had to write was typically changes to the sequential driver program to use the parallel framework, and/or changes necessary due to the use of the sequential hook methods. The extreme case of this is for the Map Overlay problem where there was a fundamental change in paradigm between the two implementations. In order to use the Pipeline pattern, the user has to create classes for the various stages of the pipeline. Each of these classes contains a specific hook method for performing the computation of that stage, and then transforming the current object to the object representing the next stage. As this was not necessary in the sequential application, the user had to write more code in order to use the Pipeline pattern.

That a significant amount of work on the part of the programmer was needed to use the Pipeline pattern demonstrates an important aspect of utility; how much must the original program be changed in order to use the tool? For the Map Overlay problem, extensive modifications were necessary to use this pattern. This point is especially important in the context of legacy code. Without knowing more about the structure of the legacy applications, this is clearly an aspect of utility that is hard to quantify.

The performance results presented in Table 3 are for a shared-memory architecture. The machine used to run the applications 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.

Table 3 shows that the use of the patterns can produce programs that have reasonable scalability. Again, these figures are not the best that can be achieved, since only the Patterns Layer of CO_2P_3S was used. All of these programs could be further tuned to improve the performance. However, the results are impressive considering that the programmer only wrote less than one hundred lines of code in most of the cases. This is possible because CO_2P_3S uses pattern templates to allow the programmer to customize the generated code for a particular application and hardware.

While most of the programs do show reasonable scalability, the two that do not, Kece and Map Overlay, are the result of application-specific factors and not a consequence of the use of the specific pattern. In the case of Kece, the number of siblings processed in parallel during the depth-first search was found to never exceed 30. If we assume that all nodes at a single level take the same amount of time to process, then the number of siblings is such that many of the processors will remain idle at each level. This is not as much of a problem at the lower levels, where the granularity of the nodes is small, as at the higher levels, 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, some nodes require a disproportionate amount of processing. This makes the problem worse as it is even more likely that processors will remain idle for a given level.

For the Map Overlay problem, the problem was only run using up to 8 processors, as the application ran for 5 s using 8 processors for the largest dataset size that the Java Virtual Machine could support.

5.5.2. Distributed-memory solutions

Table 4 shows the code metrics for using CO_2P_3S to generate distributed-memory code. As the distributed implementations of the Pipeline and Search-Tree patterns have not been done, only a subset of the problems are shown. As with Table 2, the first two columns show the size of the sequential and distributed-memory programs. The remaining three columns show the amount of code that is generated by CO_2P_3S, reused from the sequential application, and written by the programmer. A key point is that although CO_2P_3S generates very different frameworks for the shared- and distributed-memory environments, the code that the user provides is almost identical. There are only two small differences.

```
    this.wavefront = new Skyline(this.height,
                                 this.width,
                                 threads,
                                 this,
                                 this);
```

(a)

```
try{
    this.wavefront = new Skyline(this.height,
                                 this.width,
                                 threads,
                                 this,
                                 this);
}catch (java.rmi.RemoteException re) {
    re.printStackTrace();
}
```

(b)

Fig. 15. Example of the minor code difference between using shared- and distributed-memory framework code in an application: (a) use of shared-memory code; (b) use of distributed-memory code.

The first difference is that the method signatures of the generated hook methods for the distributed environment may contain a `throws` clause. For example, in the skyline matrix solver application, the signature of one of the hook methods for the shared memory environment is `operateLeft(...)`. In the distributed memory environment, the signature becomes `operateLeft(...) throws java.rmi.RemoteException`. In the distributed memory case, if an exception occurs due to a node failure, the framework code catches the exception and displays an error. Note that the user fills in exactly the same code for the hook methods in both cases. Therefore, no user code changes are required to move from one environment to the other.

The second difference is that in the distributed memory environment, the user must use a `try-catch` statement to enclose the constructor of the object that initiates the parallel computation. Fig. 15 shows an example of the shared- and distributed-memory versions of this statement for the skyline matrix solver application. It is impossible to absorb this difference into the generated framework code since the user can write code that initiates a parallel pattern from anywhere in their application code.

Two important points are demonstrated by Fig. 15 and Table 4. The first is that a user can switch between shared and distributed memory implementations by one trivial change in their application code. We are unaware of any other high-level parallel programming system that supports both shared- and distributed-memory environments in such a transparent manner. The second is how much effort the programmer saves by using the tool. For these four applications 85–95% of the final application code is generated by CO$_2$P$_3$S. Writing the same program in MPI would have taken the programmer days to weeks to produce a correct program instead of a few hours.

Table 5 shows the performance results for the distributed implementation of the Reaction/Diffusion problem. The result of just using the Pattern Layer of CO$_2$P$_3$S is shown in the first column. While these results appear poor, given the minimal amount of effort required to create the program, this is a significant result. The adjacent column shows the performance after optimizations were made to the generated Pattern Layer code at the Native Code Layer. The performance results using the Pattern Layer for the other three problems from Table 4 are not shown as they were found to be similar.

6. 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$_2$P$_3$S, provide a way to alleviate this difficulty. The CO$_2$P$_3$S 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$_2$P$_3$S system was tested by implementing the Cowichan Problem Set. This required the addition of two new patterns to CO$_2$P$_3$S, the Wavefront pattern and the Search-Tree pattern. The addition of these patterns highlight the extensibility of CO$_2$P$_3$S, an important contribution to a system's utility.

Creating shared and distributed memory solutions to the Cowichan Problems is not the only place that the CO$_2$P$_3$S system has been used. CO$_2$P$_3$S has also been used to assist in building both a web (http) server and an ftp server [31]. While this is additional evidence of the utility of CO$_2$P$_3$S, describing this problem and its solution would not have contributed significantly more to our claims.

The Cowichan Problems are 10 years old. They are an excellent representative sample of commonly occurring parallel structures. However, while measuring utility, performance cannot be ignored, and the Cowichan Problems have not scaled well to modern architectures. We challenge the community to develop a comprehensive set of applications for measuring the utility of high-level parallel programming tools. Without such a benchmark, it will be very difficult for any high level tool to prove its value and gain high acceptance.

Parallel computing must eventually move away from MPI and OpenMP. High-level abstractions have been researched for years. The most serious obstacles—performance, utility, and extensibility—are all addressed by CO$_2$P$_3$S. Another way in which the utility of CO$_2$P$_3$S is demonstrated is by the MetaCO$_2$P$_3$S tool, which produces an XML description of the patterns so that patterns may be made available to all for use in other tools through a pattern repository.

The CO$_2$P$_3$S tool is publicly available for downloading at http://www.cs.ualberta.ca/~systems/cops.

Acknowledgments

We thank iCORE and NSERC for their support of this project. We also thank Steve MacDonald who developed the initial CO_2P_3S system and Steve Bromling who created the MetaCO_2P_3S tool.

References

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

[2] J. Anvik, Asserting the utility of CO_2P_3S using the Cowichan Problems, Master's Thesis, Department of Computing Science, University of Alberta, 2002.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, P3L: A structured high level programming language and its structured support, Concurrency Practice Exp. 7 (3) (1995) 225–255.

[4] H.E. Bal, A.S. Tanenbaum, M. Frans Kaashoek, Orca: A language for parallel programming of distributed systems, IEEE Trans. Software Eng. 18 (3) (March 1992) 190–205.

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

[6] P. Boncz, Parallelizing the crossword generation game in Orca, Student Project Report, Vrije Universiteit Amsterdam, May 1994.

[7] D.S. Bouman, Parallelizing a skyline matrix solver using Orca, Student project Report, Vrije Universiteit Amsterdam, August 1995.

[8] S. Bromling, Meta-programming with parallel design patterns, Master's Thesis, Department of Computing Science, University of Alberta, 2002.

[9] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, K. Tan, Pattern-based parallel programming, Proceedings of the 2002 International Conference on Parallel Processing, August 2002, pp. 257–265.

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

[11] M. Cole, Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computations, MIT Press, Cambridge, MA, 1988.

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

[13] R.S. de Boer, Parallel thinning and skeletonization using Orca, Student Project Report, Vrije Universiteit Amsterdam, August 1994.

[14] G. Dutton, (Ed.), Harvard Papers on Geographic Information Systems: vol. 6—Spatial Algorithms: Efficency in Theory and Practice, Laboratory for Computer Graphics and Spacial Analysis, 1978.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[16] D. Goswami, A. Singh, B.R. Priess, Architectural skeletons: The re-usable building-blocks for parallel applications, Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), 1999, pp. 1250–1256.

[17] P. Langendoen, Parallelizing the polygon overlay problem using Orca, Student Project Report, Vrije Universiteit Amsterdam, August 1995.

[18] S. MacDonald, D. Szafron, J. Schaeffer, Rethinking the Pipeline as Object-Oriented States with Transformations, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'2004) at IPDPS, April 2004, Santa Fe, U.S., pp. 12–21.

[19] B.L. Massingill, T.G. Mattson, B.A. Sanders, A pattern language for parallel application programs, Proceedings of the Sixth Eurpeoan Conference on Parallel Computing (Euro-Par 2000), 2000, pp. 678–681.

[20] P. Newton, J.C. Browne, The CODE 2.0 graphical parallel programming language, Proceedings of the Sixth ACM International Conference on Supercomputing, 1992, pp. 167–177.

[21] D. Nicolaas, Parallelizing the Turing ring using Orca, Student Project Report, Vrije Universiteit Amsterdam, August 1994.

[22] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice-Hall, Englewood Cliffs, NJ, 1995 (Chapter 5).

[23] J. Schaeffer, D. Szafron, G. Lobe, I. Parsons, The Enterprise model for developing distributed applications, IEEE Parallel Distrib. Technol. 1 (3) (1993) 85–96.

[24] M. Schuetze, J.P. Riegel, G. Zimmermann, A pattern-based application generator for building simulation, Proceedings of the Sixth European Software Engineering Conference (ESEC'97), Lecture Notes in Computer Science, Springer, Berlin, 1997, vol. 1301, pp. 468–482.

[25] A.R. Sukul, Parallel implementation of an Active Chart parser in Orca, Student Project Report, Vrije Universiteit Amsterdam, August 1995.

[26] K. Tan, Supporting pattern-based parallel programming in a distributed-memory environment, Master's Thesis, Department of Computing Science, University of Alberta, 2002.

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

[28] A.M. Turing, The chemical basis of morphogenesis, Trans. Roy. Soc. London 237 (1952) 37–42.

[29] G.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, April 1994, pp. 183–193.

[30] G.V. Wilson, H.E. Bal, An empirical assessment of the usability of Orca using the Cowichan Problems, IEEE Parallel Distrib. Technol. 4 (3) (1996) 36–44.

[31] G. Zhuang, J. Schaeffer, D. Szafron, P. Earl. Using Generative Design Patterns to Develop Network Server Application, 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'2005), 2005, pp. 178.

John Anvik received a Masters degree in Computer Science from the University of Alberta in 2002. His work examined the use of generative design patterns for shared-memory parallel programs. Following his Masters degree, John worked on a bioinformatics research project and taught undergraduate courses at the University of Alberta. He is currently a PhD student at the University of British Columbia working in the Software Practices Lab.

Jonathan Schaeffer is a professor of Computing Science at the University of Alberta. He is a Canada Research Chair and an iCORE Chair. His research interests are in artificial intelligence and parallel/distributed computing. He is best known for his work on computer games. He is the creator of the checkers program Chinook, the first program to win a human world championship in any game. He is a co-founder of BioTools (bioinformatics software and the popular Poker Academy) and Chenomx (medical diagnostic software).

Duane Szafron is a Professor of Computing Science and Vice Dean of the Faculty of Science at the University of Alberta. He has been doing object-oriented computing research since 1980, including language design, language implementation, programming environments and parallel computing. He is also doing research in bioinformatics and computer games. He teaches object-oriented computing courses to students at all levels, from first year through graduate school. He is one of the founders of two University of Alberta spin-off companies: BioTools, a developer of bioinformatics software and the popular Poker Academy poker software, and Chenomx, a developer of medical diagnostic software.

Kai Tan received a Masters degree in Computer Science in 2002. He did work on pattern-based distributed and parallel programming and developed a distributed environment for the CO2P3S system. He currently works as a software developer for a storage virtualization company in Edmonton, Alberta.