Why Not Use a Pattern-based Parallel Programming System?

John Anvik, Jonathan Schaeffer, Duane Szafron, and Kai Tan

University of Alberta

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 reducing 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 a parallel program.
Using the Cowichan Problems, this paper 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. Code metrics and performance results are presented for the various applications to show the usability of the CO_2P_3S system and its ability to reduce programming effort, while producing programs with reasonable performance. Finally, the *extensibility* of CO_2P_3S is illustrated by describing how a new pattern, called the Search-Tree pattern, was added to CO_2P_3S in order to solve two of the Cowichan Problems.

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 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 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 user 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.

However, there is hope. In sequential programming there exist strategies which may be used across many problems. These strategies are called *design patterns* [6] 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, a deeper understanding of the solution to the problem is immediately conveyed and certain design decisions are implicitly made.

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 [5, 7], or templates [10]. Examples of common parallel design patterns are the fork/join model, pipelines, meshes, and work piles.

Although there have been many attempts to build pattern-based high-level parallel programing tools, 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 in a given tool is quite limited, and if the application does not match the provided patterns, then the tool is effectively useless. Further, a tool may only be suitable for a single type of parallel architecture.
3. Extensibility. High-level tools contain a fixed set of patterns and the tool cannot be extended to include more.

The CO_2P_3S parallel programming system uses design patterns 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 an application, and used to generate a parallel framework tailored for the 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 [4]. The intent of this paper 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 minimum effort, where effort is measured by the number of additional lines of code written by the CO_2P_3S user. The Cowichan Problems are used to demonstrate this utility by showing the breadth of applications which can be written using the tool. Furthermore, it is shown that the a shared-memory application can be recompiled to run in a distributed memory environment with no changes to the code.

First, a description of the CO_2P_3S system is given in Section 2. The results of using the system to implement solutions to the Cowichan Problems are then presented in Section 3. Section 4 illustrates the extensibility of CO_2P_3S by describing a new pattern, the Search-Tree pattern, that was necessary in order to solve two of the Cowichan problems. The use of CO_2P_3S is illustrated by showing how this pattern was used to implement an IDA* search application. Finally, some concluding remarks are made in Section 5.

2 The CO_2P_3S Parallel Programming System

The CO_2P_3S[1] parallel programming system is a tool for implementing parallel programs in Java [8]. 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 parameterized family of design solutions. Members of the solution family are selected based upon the values of the parameters 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 parameterization of its patterns.

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 method names in the pattern framework which are provided by the user. *Design parameters* are pattern parameters which affect the overall parallel structure of the generated framework. *Performance parameters* introduce optimizations that may improve performance by changing the internal framework code. However these changes are not visible to the user. *Verification parameters* allow for the inclusion of pieces of code in the framework to ensure its proper use and find errors in user 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 se-

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

quential *hook methods* in the framework code. 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 at the pattern level. However, due to the layered model of CO$_2$P$_3$S [8], the user has access to lower abstraction layers when necessary in order to tune the application.

Extensibility of a programming system supports increased utility. CO$_2$P$_3$S improves its utility by allowing new pattern templates to be added to the system using the MetaCO$_2$P$_3$S [3] tool. Pattern templates added through MetaCO$_2$P$_3$S are indistinguishable in form and function from those already contained in CO$_2$P$_3$S. This allows CO$_2$P$_3$S to adapt to the needs of the user; if CO$_2$P$_3$S lacks the necessary pattern for a problem then MetaCO$_2$P$_3$S supports its rapid addition to CO$_2$P$_3$S.

The descriptions of pattern templates generated by MetaCO$_2$P$_3$S are stored in system-independent XML[2] format. This ensures that the patterns generated by MetaCO$_2$P$_3$S can be used not only by the CO$_2$P$_3$S system itself, but also by any template-based programming tool which uses XML. The creation of a system-independent pattern repository can enhance the the utility of all systems that can use this format since more patterns can be developed and distributed.

The first step in testing the utility of the CO$_2$P$_3$S system was to select a suitable set of problems to use. The Cowichan Problems were chosen as a non-trivial set of problems. When the problems were analyzed, it became evident that CO$_2$P$_3$S lacked the necessary patterns to implement four of these problems. For another high-level programming system the experiment would have been over. However, using MetaCO$_2$P$_3$S, we were able to extend CO$_2$P$_3$S to fit our requirements through the addition of two new patterns: the Wavefront pattern [1, 2] and the Search-Tree pattern.

3 Using CO$_2$P$_3$S to Implement the Cowichan Problems

Test suites such as SPEC and SPLASH for assessing system *performance*, 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 [12]. 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 [13]. 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.

The 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-grained

[2] Extensible Markup Language.

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. Descriptions of each of these problems can be found in [12].

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. 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. Therefore, a different single-agent search (IDA*), which was more representative of this class of problems was selected.

All of the Cowichan Problems have been implemented using CO_2P_3S. Specifics of how the patterns were used to implement the problems may be found in [2]. Presented here is an overview of each of the patterns used to solve the problems. Table 1 provides a summary of which CO_2P_3S pattern was used to solve each of the Cowichan Problems. Tables 2 and 3 provide code metrics and performance results for each solution that was created using CO_2P_3S.

3.1 The Search-Tree Pattern

The Search-Tree pattern is used to parallelize tree search algorithms, such as those used in 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).

3.2 The Wavefront Pattern

The Wavefront pattern [1, 2] 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. 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. CO_2P_3S contains versions of the Wavefront pattern for both shared-memory [2] and distributed memory architectures [11].

3.3 The Mesh Pattern

The Mesh pattern [8] is used for computing elements of a regular, rectangular two-dimensional data set where each element is dependent on its surrounding values and changed 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 from either the cardinal points or from 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 [8] and distributed memory implementations of the Mesh pattern [11].

3.4 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 transition between pipeline stages is simply a change of state for the object [8].

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 [8] in CO_2P_3S 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 is balanced across the pipeline.

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 1. Patterns used to solve the Cowichan Problems.

3.5 The Effects of Using CO_2P_3S

The results of using CO_2P_3S to implement solutions to the Cowichan Problems are presented here. The 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 [8].

The results are presented in two tables. Table 2 shows the code metrics from the various implementations and contains the sizes of the sequential and parallel programs, how much of the parallel code was generated by CO_2P_3S, how much code was reused from the sequential application, and how much new sequential code the user was required to write. Table 3 provides performance results for various sets of processors.

Table 2 shows that a sequential program can be adapted to a shared memory parallel program with little additional effort on the part of the user. 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. The additional code that the user was required 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 Overly problem where there was a fundamental change in paradigm between the two implementations. In order to use the Pipeline pattern, the user is required to create classes for various stages of the pipeline. Each of these classes is required to contain a specific hook method for performing the computation of that stage, and for transforming the current object to the object representing the next stage. As this was not necessary in the sequential application, the user was required to write more code in order to use the Pipeline pattern. For all the other patterns, the user only had to fill in the hook methods for a generated class.

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 pattern layer of CO_2P_3S was used. All of these programs could be furthered tuned to improve the performance. 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 exceeded 20. For the Map Overlay problem, the problem was only run using up to 8 processors, as

Application	Sequential	Parallel	Generated	Reused	New
Fifteen Puzzle	125	308	123	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 2. Code metrics for the shared-memory implementations of solutions to the Cowichan Problems.

Application	2	4	8	16
Fifteen Puzzle	1.74	3.56	6.70	10.60
Kece	1.93	3.42	4.83	5.80
Skyline Matrix Solver	1.93	3.89	7.84	14.86
Matrix Product Chain	1.81	3.64	7.80	13.37
Map Overlay	1.56	3.11	4.67	-
Image Thinning	1.88	3.53	6.39	10.43
Reaction/Diffusion	1.75	3.13	4.92	6.50

Table 3. Speedups for the shared-memory implementations of solutions to the Cowichan Problems.

the application ran for 5 seconds using 8 processors for the largest dataset size that the JVM[3] could support.

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 are not yet complete, only a subset of the problems are shown. 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.

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 second case, if an exception occurs due to a node failure, the framework code catches the exception and displays an error. Note that 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. Figure 1 shows an example of the shared-memory

[3] Java Virtual Machine

Application	Sequential	Parallel	Generated	Reused	New
Skyline Matrix Solver	196	1929	1760	144	25
Matrix Product Chain	68	1534	1458	60	16
Image Thinning	221	2138	1968	170	12
Reaction/Diffusion	263	1476	1304	177	55

Table 4. Code metrics for the distributed-memory implementations of solutions to the Cowichan Problems.

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. The important point 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 memory and distributed memory environments in such a transparent manner. Since work on the distributed memory patterns is ongoing, no performance results are shown in this paper. However, we have generated working distributed-memory versions of the Wavefront and Mesh applications from the Cowichan Problems.

4 The Search-Tree Pattern

Two new patterns were added to CO_2P_3S to solve the Cowichan Problems. The Wavefront pattern has been described in [1]. The Search-Tree pattern was also added to CO_2P_3S to solve two of the Cowichan Problems. Using the MetaCO_2P_3S tool, the CO_2P_3S system was extended to support this new pattern. Once the pattern had been designed, adding it to CO_2P_3S took approximately nine hours. This demonstrates the extensibility of CO_2P_3S, which contributes to the system's utility.

4.1 Pattern Parameters

The single lexical parameter for the Search-Tree pattern is the name of the class which represents a node in the tree. This class will contain the hook methods which are implemented by the pattern user.

This 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. 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 2 shows the order in which nodes are processed for both breadth-first and depth-first parallel searches. Another possible traversal is best-first. However we add parameters and parameter values to CO_2P_3S on a need-only basis and we do not yet have an application that needs this traversal. We take this approach to prevent the generation of an overly general framework or the unnecessary explosion of parameter combinations.

The Search-Tree pattern has one performance parameter, *early termination*. This parameter 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 parameter type to the CO_2P_3S system called a *verification* parameter. In the Search-Tree pattern, the verification parameter 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 is still waiting for the completion of its children, but the framework can detect that in fact all 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 waiting for more child nodes to be processed when they have all been processed.

4.2 Pattern Hook Methods

The parallel pattern framework generated by CO_2P_3S for the Search-Tree pattern contains five hook methods into which the user inserts their sequential code. Depending on the parameter settings, two additional framework methods may be generated which the user can make use of in their code. This is demonstrated in Section 4.4. 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 methods are:

`divideOrConquer()` This hook method indicates whether to generate a node's children (i.e. call `divide()`), which will then be processed in parallel or to proceed with the sequential computation of the node (i.e. call `conquer()`).
`divide()` This hook method generates a node's children.
`conquer()` This hook method performs the sequential computation of a node.
`updateState(TreeNode child)` This hook 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 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 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 Figure 3. 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 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 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 Figure 3.

Note that while there are many tools which could produce code like that shown in Figure 3, CO_2P_3S uses generative design patterns so that only the portions of the code relevant to the selected traversal method and other parameter 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. Figure 4 shows how updates are propagated up the tree and node completion is verified.

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 parameter values, have CO_2P_3S generate the customized parallel framework code, and implement the necessary hook methods.

4.4 Implementation of IDA* Search

This section provides an example of how CO_2P_3S is used to generate a parallel program from a sequential program. Specifically, the IDA* search used in the Fifteen Puzzle [9] is implemented 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 from the CO_2P_3S perspective this is a breadth-first search, but that the subtrees which are searched sequentially each perform depth-first searches. As soon as a solution is found the search should terminate, so the *early termination* parameter is set to `true`. Figure 5 shows the parameterization of the Search-Tree pattern for the Fifteen Puzzle application.

The 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. 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. If the *early termination* parameter is set, then two framework methods are provided to the user: `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 `update()` method collects the nodes in the upper portion of the tree which are on the path toward the goal state.

In this application, all of the 125 lines were reused from the sequential application 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.

5 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 and before it can gain wide user acceptance there needs to be a confidence that the tool can provide the assistance necessary. To this end, the utility of the CO_2P_3S system was tested by implementing the Cowichan Problem Set. This required the addition of a two new patterns to CO_2P_3S, the Wavefront pattern and the Search-Tree pattern. The addition of these patterns highlight the extensibility of CO_2P_3S; an important contribution to a system's utility.

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_2P_3S. Another way in which the utility of CO_2P_3S is demonstrated is by the MetaCO_2P_3S tool, which produces XML description of the patterns so that patterns may be made available to all for use in other tools through a pattern repository.

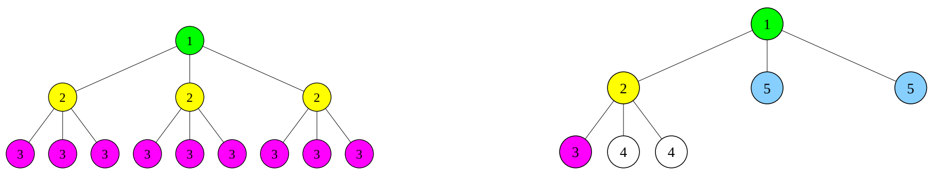
```
this.wavefront = new Skyline(this.height,
                                this.width,
                                threads,
                                this,
                                this);
```

(a) Use of shared-memory code.

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

(b) Use of distributed-memory code.

Fig. 1. Example of the minor code difference between using shared-memory and distributed-memory framework code in an application.



(a) Breadth-first traversal. (b) Depth-first traversal.

Fig. 2. Tree traversals in the Search-Tree pattern. Nodes with the same value are processed in parallel.

```
if(node is invalid) return

if(divideOrConquer())
    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
```

Fig. 3. Pseudo-code of the `process()` method.

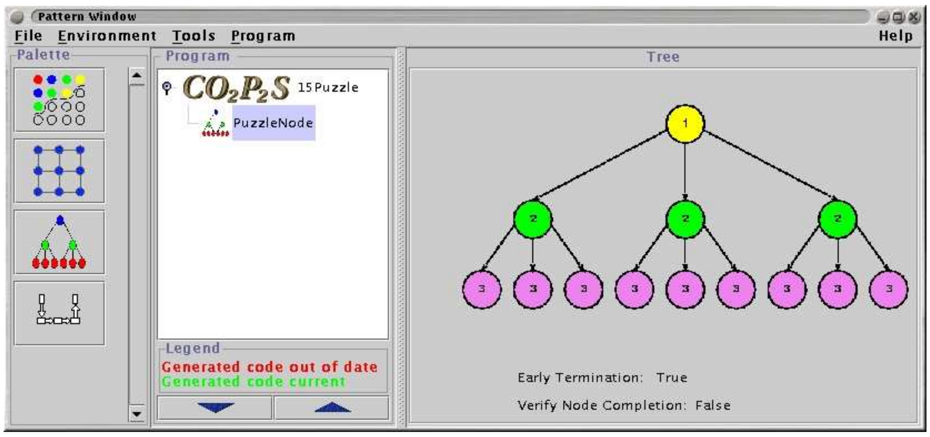
```
update state
remove child from validation list

// 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
```

Fig. 4. Pseudo-code of the `update()` method.



Early Termination: True

Verify Node Completion: False

Fig. 5. The parameterization of the Fifteen Puzzle in CO_2P_3S.

References

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. On CD.
2. John K. Anvik. Asserting the utility of COPS using the Cowichan Problems. Master's thesis, Department of Computing Science, University of Alberta, 2002.
3. Steve Bromling. Meta-programming with parallel design patterns. Master's thesis, Department of Computing Science, University of Alberta, 2002.
4. Steve Bromling, Steve MacDonald, John Anvik, Jonathan Schaeffer, Duane Szafron, and Kai Tan. Pattern-based parallel programming. *Proceedings of the 2002 International Conference on Parallel Processing*, August 2002.
5. Murry Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computations.* MIT Press, 1988.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
7. 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.
8. Steve MacDonald. *From Patterns to Frameworks to Parallel Programs.* PhD thesis, Department of Computing Science, University of Alberta, 2001.
9. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 5. Prentice Hall, 1995.
10. 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.
11. Kai Tan. Supporting pattern-based parallel programming in a distributed-memory environment. Master's thesis, Department of Computing Science, University of Alberta, 2002.
12. 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.
13. 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.