# Why Not Use a Pattern-based Parallel Programming System?

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

Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{janvik, jonathan, duane, cavalier}@cs.ualberta.ca

**Abstract.** Parallel programming environments provide a way for users 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, using a pattern-based approach to express concurrency. This paper demonstrates that the $CO_2P_3S$ system contains a rich set of parallel patterns for implementing a wide variety of applications running on shared-memory or distributed-memory hardware. Code metrics and performance results are presented to show the usability of the $CO_2P_3S$ system and its ability to reduce programming effort, while producing programs with reasonable performance.

## 1 Introduction

In sequential programming, a powerful paradigm has emerged to simplify the design and implementation of programs. *Design patterns* encapsulate the knowledge of solutions for a class of problems [4]. 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 [3, 5], or templates [7]. 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 programming 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 limited, and if the application does not match the provided patterns, then the tool is effectively useless.
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* [6], 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-memory and distributed-memory environments.
3. Extensibility. Meta$CO_2P_3S$ is a tool used for rapidly creating and editing $CO_2P_3S$ patterns [2]. $CO_2P_3S$ currently supports 17 parallel and sequential design patterns, with more patterns under development.

This paper focuses on the utility aspect of $CO_2P_3S$. The 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 minimum effort, where effort is measured by the number of additional lines of code written by the $CO_2P_3S$ user. The Cowichan Problems [8] 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 code. We know of no other parallel programming tool that can support the variety of applications that $CO_2P_3S$ can, is extensible, supports both shared-memory and distributed-memory architectures, and achieves good performance for the user effort expended.

## 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 [6]. $CO_2P_3S$ 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

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

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.

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*. This abstraction 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 $CO_2P_3S$ [6], the user has access to lower abstraction layers when necessary in order to tune the application.

Extensibility of a programming system supports increased utility. $CO_2P_3S$ improves its utility by allowing new pattern templates to be added to the system using the Meta$CO_2P_3S$ tool [2]. Pattern templates added through Meta$CO_2P_3S$ 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 Meta$CO_2P_3S$ supports its rapid addition to $CO_2P_3S$.

## 3 Using $CO_2P_3S$ to Implement the Cowichan Problems

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 [8, 9], a suite of seven problems specifically designed to test the breadth and ease of use of a parallel programming tool.[2]

When the Cowichan problems were analyzed, it became evident that $CO_2P_3S$ lacked the necessary patterns to implement four of these problems. For any other high-level programming system, the experiment would have been over. However, using Meta$CO_2P_3S$, we were able to extend $CO_2P_3S$ to fit our requirements through the addition of two new patterns: the Wavefront pattern and the Search-Tree pattern [1]. For each of these patterns, simple parameterization made them general enough to handle a wide class of applications.

Table 1 provides a summary of which $CO_2P_3S$ pattern was used to solve each of the Cowichan Problems. Note that while $CO_2P_3S$ supports using multiple patterns in an application, this capability was not needed here.

## 4 Evaluating $CO_2P_3S$

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

[2] One modification was made to the original problem set. The single-agent search problem (Active Chart Parsing) takes only a few seconds of CPU time on a modern processor. Therefore, a different single-agent search (IDA*), which was more representative of this class of problems, was used.

**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 for the shared-memory implementations.

| 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 |

with minimal user effort, reasonable speedups can be achieved. The speedups are not necessarily the best, since the applications could be further tuned to improve performance using the $CO_2P_3S$ layered model [6].

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 a shared-memory computer.

Table 2 shows that a sequential program can be adapted to a shared-memory parallel program with little additional effort on user's part. 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 (if a new pattern had to be created) 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. 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 had to write more code to use the Pipeline pattern. For all

**Table 3.** Speedups for the shared-memory implementations.

| 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 4.** Code metrics for the distributed-memory implementations.

| 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 |

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 GB 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; all of these programs could be furthered tuned to improve the performance. While most of the programs 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 20, and was usually less than 10 resulting in processors being starved for work. For the Map Overlay, the problem was only run using up to 8 processors, as the application ran for 5 seconds using 8 processors for the largest dataset size that the JVM 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 have not been done (a lack of resources), 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 one-line differences (to handle distributed exceptions).

## 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 two new patterns to $CO_2P_3S$, highlighting 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 $CO_2P_3S$.

$CO_2P_3S$ is available for download at `http://www.cs.ualberta.ca/~systems/cops/index.html`. A longer version of this paper is available at `http://www.cs.ualberta.ca/~jonathan/Papers/par.2003.html`.

## Acknowledgments

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

## References

1. J. Anvik. Asserting the utility of COPS using the Cowichan Problems. Master's thesis, Department of Computing Science, University of Alberta, 2002.
2. S. Bromling. Meta-programming with parallel design patterns. Master's thesis, Department of Computing Science, University of Alberta, 2002.
3. M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computations.* MIT Press, 1988.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
5. D. Goswami, A. Singh, and B. Priess. Architectural skeletons: The re-usable building-blocks for parallel applications. In *Parallel and Distributed Processing Techniques and Applciations (PDPTA'99)*, pages 1250–1256, 1999.
6. S. MacDonald. *From Patterns to Frameworks to Parallel Programs.* PhD thesis, Department of Computing Science, University of Alberta, 2001.
7. J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.
8. G. Wilson. Assessing the usability of parallel programming systems: The Cowichan problems. In *IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, 1994.
9. G. Wilson and H. Bal. An empirical assessment of the usability of Orca using the Cowichan problems. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.