

Is MPI Suitable for a Generative Design-Pattern System?

Paras Mehta, José Nelson Amaral, Duane Szafron
Department of Computing Science
University of Alberta, Edmonton, Alberta, Canada

Abstract

Generative parallel design patterns is a proven technique to improve the productivity of parallel program development. However many of the generative design-pattern systems are developed for target languages that are not widely used by the high performance computing community. This paper describes an initial effort to develop a system that will hopefully answer the question in the title in the affirmative. This new system is ostensibly based on, and build upon the experience with, the successful CO₂P₃S. Significant challenges must be overcome to implement the features of a system that generates frameworks conceived for an object-oriented programming language (Java) into a parallel-annotated procedural language (MPI/C).

1 Introduction

This paper describes preliminary investigation and experimentation towards the development of the MPI Advanced Pattern-Based Parallel Programming System (MAP₃S), a generative pattern system that is an evolution of the Correct Object-Oriented Pattern-Based Parallel Programming System (CO₂P₃S) [8]. CO₂P₃S uses *pattern templates* to represent a family of frameworks that solve a problem. CO₂P₃S is a successful demonstration that generative design patterns can be used to generate solutions to parallel programming problems. The parallel programs generated by CO₂P₃S exhibit significant speed improvements over corresponding sequential versions.

The target programming language of CO₂P₃S is multi-threaded Java. This choice of language provides several of the advantages of object-orientation, such as encapsulation and polymorphism, that facilitate the integration of the framework generated by CO₂P₃S with the code written in hook methods by the pattern user. However Java is not yet broadly used in the High-Performance Computing (HPC) community and many implementations of Java still produce code that is slower than parallel-annotated procedural language equivalents.

On the other hand the Message Processing Interface (MPI) standard is broadly used, has a significant

installed base, and is known to many HPC programmers. Thus, an interesting question is: Can the programming-productivity gains demonstrated by CO₂P₃S be duplicated in a system that targets MPI? Such a system would provide for a fast dissemination of the use of design patterns in the HPC community. This is the motivation for MAP₃S.

2 Multithreaded Java vs. MPI

Hierarchical design, encapsulation, polymorphism, and thread objects make multi-threaded Java very suitable for pattern-based programming. Some of these features have to be emulated in MPI/C to enable the implementation of generative design patterns. In the original shared-memory version of CO₂P₃S, data communication between threads is performed by an underlying library [2]. A distributed-memory version of CO₂P₃S required only changes to the lower-level communication code [12]. One of the main contributions of CO₂P₃S is the separation of parallelism concerns from computation concerns. Parallelism concerns, such as synchronization and communication, become the responsibility of the pattern designer, while computation concerns become the responsibility of the pattern user.

MPI requires function calls to transfer data between processes. In MPI the distribution of tasks to processors requires the exchange of messages. Moreover, data transfers require the matching of sends and receives between the processes involved. Thus the separation of parallelism and computation concerns in MAP₃S is more challenging than in CO₂P₃S. This paper describes the experience of translating two patterns, a mesh and a search-tree, from CO₂P₃S to MAP₃S to illustrate some of the difficulties encountered and the solutions adopted.

Despite the significant differences between multi-threaded Java and MPI/C, MAP₃S should maintain CO₂P₃S's ability to generate framework code from user-chosen patterns and pattern parameters. Also, MAP₃S must implement most, or all, of the patterns available in CO₂P₃S. And despite the greater challenge of accomplishing it in MPI when compared with Java, segregating parallelism from computation remains a priority.

An important characteristic of CO₂P₃S that MAP₃S should maintain is the ability to tune the generated framework code to specific problems through the use of pattern parameters. Giving the pattern user the ability to adapt the generated code in order to best utilize both problem characteristics and the underlying architecture is important in a performance-driven pattern-based programming system.

In MAP₃S the user-defined sections of code are preprocessor macros that are inserted into the proper place in the framework — instead of the hook methods of CO₂P₃S. Because the code in these macros is inserted into functions in the framework, there is a potential for conflicting use of local scoped variables and for semantic overwriting. Thus, the pattern user must exercise greater care when writing the code for MAP₃S's macros than for CO₂P₃S's hook methods. On the other hand macro code is automatically inlined

into the framework code, and thus does not incur the function call overhead of a hook method.

3 Patterns

When selecting patterns for the initial investigation of MAP₃S, the goal was to select patterns that are quite distinct. The mesh pattern and the search-tree pattern were selected as the initial targets for MAP₃S, since they exhibit the most differences of all patterns. A mesh is a data-parallel pattern and a search tree is a task-parallel pattern. Moreover, very distinct strategies for the implementation of the patterns were taken. The mesh pattern implements a static load distribution very similar to the one in CO₂P₃S. The search-tree pattern implements a work-stealing dynamic load balancing mechanism that is common-place in multi-threaded systems, but was never used in CO₂P₃S. While describing the experience with the implementation of these patterns, this section highlights the differences between CO₂P₃S and MAP₃S.

3.1 Mesh Pattern

Problems involving multi-dimensional arrangements of elements for which the computation of one element is dependent on the element's neighbors are good candidates for the Mesh Pattern. In a typical mesh problem the value of an element e at time $t + 1$ is computed based on the value of e and the neighbors of e at time t . New values for all the elements of a region of the mesh may be computed for several iterations before the values at the edges of the region are transmitted to neighboring regions. This process continues until some condition holds true. For example, a fixed number of iterations is executed, or a stability condition is reached.

3.1.1 Mesh decomposition and synchronization

Currently MAP₃S implements a two-dimensional mesh with the elements organized into a two-dimensional array. Each array element is a data structure defined by the pattern user. The pattern breaks down this two-dimensional array into blocks. The arrangement of these blocks is also defined as a smaller two-dimensional array. For instance, a 2000×2000 mesh can be formed by 200×100 blocks. Thus the initial mesh is represented by a 10×20 array of blocks. An element that has at least one neighbor that is not in the same block is a *fringe element*.

After initialization, these blocks are distributed to the various processors before the computation proceeds. Synchronization points (*e.g.*, barriers) are placed between iterations. At these points, processors computing neighboring blocks swap their respective fringe elements. Once the termination condition is reached, the final blocks are collected by a single process, and results are computed from the reintegrated mesh structure.

MAP₃S has to comply with MPI message size restriction. It is often not possible to transfer entire rows and columns between processes in a single message. On the other hand, transferring single elements is extremely inefficient. Thus, a maximum packet size, expressed in bytes, is defined for the program. At synchronization points, packets will contain as many elements as allowed under the specified packet size.

3.1.2 Pattern Parameters

MAP₃S maintains the customization parameters provided by CO₂P₃S for the mesh pattern, including the number of neighbours (cardinal neighbours vs. all eight), horizontal and vertical toroidality (whether and in what directions the mesh wraps around), and a synchronization parameter (whether each region in the mesh waits for new neighboring values before going to the next iteration or not).

MAP₃S implements additional parameters that allow the user to better tune the code for specific architectures. Currently the mesh-pattern user can specify the following parameters:

Packet size. The maximum size of a packet transmitted between neighboring processes. The generated framework will break fringe columns and rows in packets that are not larger than the packet size specified by the pattern user.

Update timing. For conceptual simplicity, synchronizing fringes between neighbours occurs between iterations; a wholesale swap is made at once between all neighbouring blocks. The cost of this simplicity, however, is potential contention for message passing. Another strategy to accomplish this synchronization would be to only communicate a fringe packet when it is requested. This parameter enables the pattern user to decide whether synchronization occurs wholesale or on demand.

3.1.3 Pattern Hooks and Element Representation

In the CO₂P₃S implementation of a mesh, the pattern user writes *hook methods*. Exterior to the iteration cycle, are the `initialize` and the `reduce` methods that take care of pre- and post-iteration processing, respectively. During the iteration cycle the methods `prepare` and `postprocess` take care of work that must be done before and after each iteration. Finally, there is an iteration method for each class of element. For instance, in a vertically toroidal mesh with only cardinal neighbours, an element falls into one of three classes: an interior element (neighbours in all four directions), a left-edge element (no left neighbour), and a right-edge element (no right neighbour). Each of these classes requires its own iteration hook method.

Elements of the mesh are represented by a C structure defined by the pattern user, rather than by Java objects in CO₂P₃S. Because the elements of the mesh are packed into MPI packets before being sent, special macros to perform this packing must be provided. Presently, these macros must be defined by the pattern user, but in the future they will be generated automatically by MAP₃S.

3.1.4 Block Distribution with Fixed Message Sizes

Because, in the MPI system, message passing is explicit, and because there are limitations on the size of message that can be passed efficiently, transmitting an entire row or column of a large sized block with large elements can be difficult in MAP₃S. The solution adopted is to allow the pattern user to customize the size of the messages that are used to communicate neighboring fringe values.

Another important issue in the translation of the CO₂P₃S mesh pattern to a MAP₃S mesh pattern is the mechanism to distribute elements to the various processors. In the CO₂P₃S mesh pattern, there is no explicit division of labour between processors; each processor is given an element and performs the computation. If the element's neighbours are not local, then data is fetched from a block being operated on by another processor.

Such open-ended work distribution in MAP₃S could generate excessive communication and lead to an inefficient implementation. Instead, MAP₃S implements a system of block distribution based on the block arrangements and on the block dimensions.

The pattern user define the overall mesh dimensions as well as the dimensions of the component blocks of the mesh.¹ The framework uses these dimensions to determine the two-dimensional arrangement of the blocks over the mesh. Next, the blocks are distributed round robin to each process in the mesh. Given the arrangement of the blocks, each process can determine the neighbours of the fringe elements of its block(s).

By distributing blocks in this manner, but defining block dimensions independently of the overall mesh, the MPI requirements of explicit communication can be satisfied. At the same time the pattern user has flexibility in terms of distribution method and block size.

3.2 Search-Tree Pattern

Tree search is a task that often results in long run times and, thus, is a desirable target for parallelization. Solutions to problems such as optimization search and adversarial search can be implemented using this pattern. For instance, in an adversarial search the states of a game can be represented as nodes in the tree.

The search goes through two phases: divide and conquer. The divide phase involves the generation of the children of a given node. When tree searching is modeled as task-based parallelism, the divide phase consists on the generation of many parallel tasks to be distributed and consumed by multiple processes.

The conquer phase executes the computation at a node and updates the value of the node's parent upon completion. Thus conquer consists of the consumption of tasks, and the state update that may arise from that consumption.

¹These dimensions can be parameters whose values are only known at runtime.

3.2.1 Stealing Children

For the purposes of parallelization, the search tree is recursively broken down into **Tasks**. Each **Task** represents the root of a subtree. A **Task** is represented by a closure containing a pointer to a function to be executed and the value of the parameters required by that function. These closures are modeled using user-defined structures, with some mandatory structure elements required by the pattern such as a pointer to the parent and a reference to the generating process.

When a **Task** is consumed by a processor, one of two things may happen depending on the current phase of the search. In the divide phase, children tasks are generated to be consumed later. Typically the divide phase continues up to a depth threshold. In the conquer phase, a task consists of solving sequentially the entire subtree rooted at the node associated with the task. Once the value of the root is determined, it is forwarded to the parent of the node and the task terminates.

At the end of the conquer phase, the parent of the node being processed needs to be updated. Often, because of work stealing, the child has been processed by a different processor than the parent. Therefore, a message from the processor that conquered the child must convey the result to the processor that processes the parent. This return of information requires an additional structure defined by the pattern user, the **Result**, which is used to communicate necessary update information to the parent.

3.2.2 Search-Tree Pattern Parameters

Currently CO₂P₃S only has a shared memory implementation of the search-tree pattern — no distributed memory version exists. The default implementation of this shared-memory search-tree pattern in CO₂P₃S uses a shared work queue to distribute tasks among a fix number of threads. A task is associated with a node in the search tree. If the node is above a pre-defined level, the task creates children tasks and adds them to the work queue. Otherwise the task sequentially executes the subtree rooted at the node. Any thread takes a task off the shared work queue, executes it and goes back to the queue for another task until the queue is empty. The pattern user may override this default implementation by providing a different implementation of the hook method, `divideOrConquer`. The parameters for the search-tree pattern in CO₂P₃S include traversal-type, early-termination, and verification.

MAP₃S implements a different dynamic-load-balancing mechanism, based on distributed work queues and work stealing. In MAP₃S each processor implements a local double-ended task queue [10]. When new tasks are generated they are added to this queue. A processor that finds that its task queue is empty sends messages to other processors to beg for work. This work-stealing dynamic-load-balancing mechanism gives origin to the following additional parameters to the search-tree pattern is MAP₃S:

Divide depth. Determines how far down the tree the divide phase goes. For an application that generates completely balanced trees, it is sufficient for the divide phase to proceed up to the point in which

there is one task for each processor. For applications where there might be great imbalance in the amount of work in each subtree, the divide phase should generate many more tasks than the number of processors to enable dynamic load balance.

Stealing. The default stealing algorithm follows a beggar’s model where the first time that a processor needs work, it randomly selects a processor to request a task from. Subsequently the processor will go back to ask for tasks from the last processor that gave it a task until that processor no longer provides tasks. At that point the beggar selects randomly again. This parameter allows the pattern user to specify that a processor should always beg from its neighbors first. Such a strategy may benefit architectures where there is locality of memory references between neighboring processors.

Queue. By default, tasks consumed by the local processor are taken from the same end of the queue where locally generated tasks are added to, and tasks are stolen from the other end. This strategy typically favors locality of reference between successive tasks. This parameter allows the pattern user to change the operation of the double-ended queue to operate it either as a single last-in-first-out stack or a single first-in-first-out queue.

3.2.3 User-defined Code

In CO₂P₃S, the `divideOrConquer` method determines the end of the divide phase. There are the self-explanatory `divide` and `conquer` methods, as well as an `updateState` method (for updating the parent of a node) and a `done` method, for determining when a node is complete.

These methods are, in general, implemented by corresponding macros in MAP₃S. An exception is that there are no divide macro and no conquer macro. Instead, the function of these pattern hooks are handled by a single `processTask` macro, with the parent updating taking place in a separate macro.

3.2.4 The Task Data Structure

In the MAP₃S search-tree pattern, each processor in the system maintains its own double-ended queue of tasks. If there are no tasks in the queue, the processor attempts to steal work from another processor. During early execution, while tasks are being generated from the root, these tasks are stolen by the other processors. Because idle processors seek out tasks, and busy processors generate more tasks, the work is quickly distributed among the processors. Moreover, if a processor finishes earlier, re-balancing of work happens automatically and without central control, a key feature in this distributed-queue work-stealing model.

A subtle difference between the CO₂P₃S and the MAP₃S implementation of the search-tree pattern is the nature of information passed from one task to another. In MPI the size of a message must be explicitly defined and must be known by the receiving processor. In order to make the pattern implementation

more modular, in MAP₃S, the message that contains a task is stored in a `Task` data structure. Thus the maximum size of each `Task` must be known at compile time. In comparison to CO₂P₃S, this requirement in MAP₃S further constrains the type of parameters that can be passed to a task.

3.2.5 Beggar’s Interruptions

A limitation in the use of the MAP₃S implementation of the search-tree pattern is the absence of a mechanism to share application-wide data among the processors. For instance, this limitation may reduce the efficiency of an alpha-beta search because it will prevent potentially important pruning of the search tree. In this case the divide phase reduces to a minimax search, with the extra overhead such a search incurs.

The implementation of the work-stealing mechanism in MAP₃S requires that a processor that is performing a sequential computation be interrupted to send a task in response to a beggar’s request. The solution was to implement the pattern using two kernel threads in each processor: one thread processes tasks and performs application specific sequential computations, while the other thread blocks on a message receive and simply processes incoming messages (work requests, sent work, termination, etc.).

MAP₃S implements these dual threads using Pthreads. Thus it has little control over the allocation of resources for these threads. Ideally, MAP₃S should assign higher priority to the communication thread to ensure that request for work from other processors are dealt with speedily. However, as far as we know, the assignment of priorities to threads in Pthreads is advisory and the ultimate priority mechanism used is left to the underlying operating system.

While this first implementation of dynamic load balancing using distributed work queues in MAP₃S was done for the search-tree pattern, a similar mechanism can be used for several other patterns. For instance, image processing problems that have varying computational complexity for various parts of the image can benefit from over-decomposing the image into many tasks, and dynamically load balancing as necessary.

4 Results

4.1 Test Problems

Problems from the suite of Cowichan problems were implemented in MAP₃S to test the efficacy of its patterns [13]. Two problems were implemented for the mesh pattern: Mandelbrot Set Generation and Conway’s Game of Life. Both of these problems involve iterations over elements of two-dimensional data structures. The Mandelbrot Set Generation is embarrassingly parallel while the Conway’s Game of Life requires synchronization, and thus tests the synchronization support built into the pattern.

The Game of Kece was used to test the Search-Tree Pattern. The Game of Kece is a two-player, zero-sum game that models crossword generation. Given an $N \times M$ board, and a single placed word, players

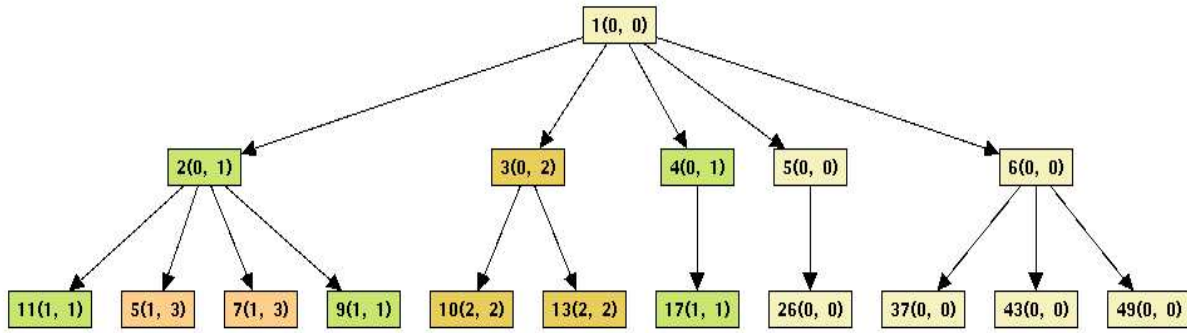


Figure 1: Dynamic load balancing in the Game Kece

must select words from a list and place them on the board. Placed words must share at least one letter with a previously placed word. The score for a move is the number of previously placed words that a given word crosses. This game uses adversarial search, such as alpha-beta search. Parallel implementations of adversarial searches may perform more work than their sequential counterparts because it may be difficult to implement tree pruning.

4.2 Usability

The MAP₃S mesh pattern provides the same code generation functionality that is available in CO₂P₃S . This flexibility allows a pattern user to tackle a similar range of problems as the problems already implemented in CO₂P₃S [2].

4.2.1 Pattern-User Code

After using the MAP₃S patterns to solve the Game of Kece and Conway’s Game of Life problems, the amount of pattern-user-written code was assessed. For the Game of Life, only 115 of 1016 lines of header file code (just over 11%) needed to be user-defined; the rest could be pattern-generated. Similarly, for the Game of Kece, user-written code makes up 67 of 606 lines of header code (again, just over 11%), indicating that the pattern-user contribution to the overall program, and hence the complications associated with creating parallel implementations of these algorithms, are significantly reduced. This ratio will not be reflected in programs requiring far more work in the pattern hook.

4.2.2 Dynamic Load Balancing

The MAP₃S implementation of tree search is likely to generate parallel programs with higher performance for problems that generate unbalanced trees. The requirement that maximum message sizes be known

when the inter-processor communication mechanism is setup may create additional hurdles to implement some tree-search problems in MAP₃S when compared with CO₂P₃S.

Figure 1 shows a small search tree generated for the game of Kece. This figure illustrates the dynamic load balancing that occurs during the expansion of a game of Kece minimax search within MAP₃S. Each node in the figure has a unique identifier and two processor identifiers for, respectively, the processor that generates the node and the processor that consumes the node. For example, the subtree rooted at node 2(0, 1) is generated by processor 0 and stolen by processor 1.

Most of the work (the sequential search) is done at the leaf level. Initially processor 0 processes the root node and generates five child nodes, three of which are stolen by other processors. Besides the good distribution of tasks at the leaf-level, this figure also indicates that the beggar's model of work stealing is preserving locality. Notice that processor 3 goes back to processor 1 to get more work when it is done with its first leaf computation. Likewise processor 1 goes back to processor 0 for more work when it is done with the first subtree that it had stolen.

4.3 Speedup

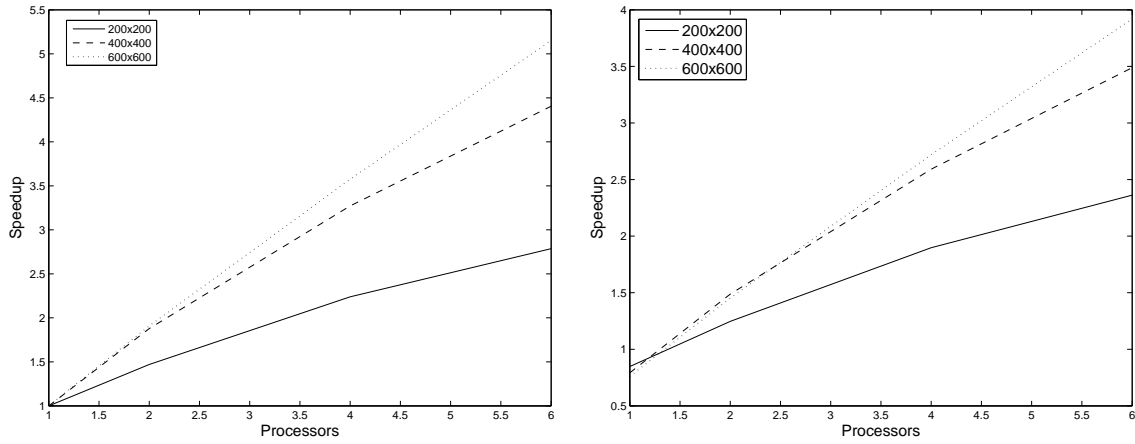
For comparisons against a sequential version of the code, the MAP₃S code was executed on an IBM p690, 12-processor machine running AIX 5.1. Sequential code was compiled using the native AIX C compiler, while MPI code was compiled using a thread-safe native MPI compiler. Both sequential and parallel versions were compiled using '-O3' optimization level. Speedup is determined by running three executions of the code under the given conditions, averaging the run times, and dividing by the baseline runtime. For comparison against a single processor execution, pattern-generated code run on a single processor is the baseline. For comparison against a sequential execution, sequential C code is used as a baseline.

4.3.1 Mesh Pattern

Speedup for the Mesh Pattern is determined from runs of the Game of Life for subsequently larger matrices. The Game is run for 4000 iterations on square matrices of dimension 200x200, 400x400, and 600x600.

Figure 2(a) shows speedup when the runtime of a multiprocessor execution of the Game of Life is compared to a single processor execution of the pattern generated code. For large enough matrices, the speedup is close to linear, but tapers away from linear as more processors are added, a common characteristic of parallel programs. Despite the drop-off as more processors are added, the pattern still attains over 5 times speedup for six processors, indicating that the pattern does manage to distribute the workload fairly well.

Figure 2(b) shows a more relevant comparison: against a sequential solution to the Game of Life. For a single processor execution, the pattern-generated code runs slower than the sequential code (as expected),



(a) Speedup against a single processor parallel version of the Game of Life

(b) Speedup against a sequential version of the Game of Life

Figure 2: Speedup attained on Conway’s Game of Life

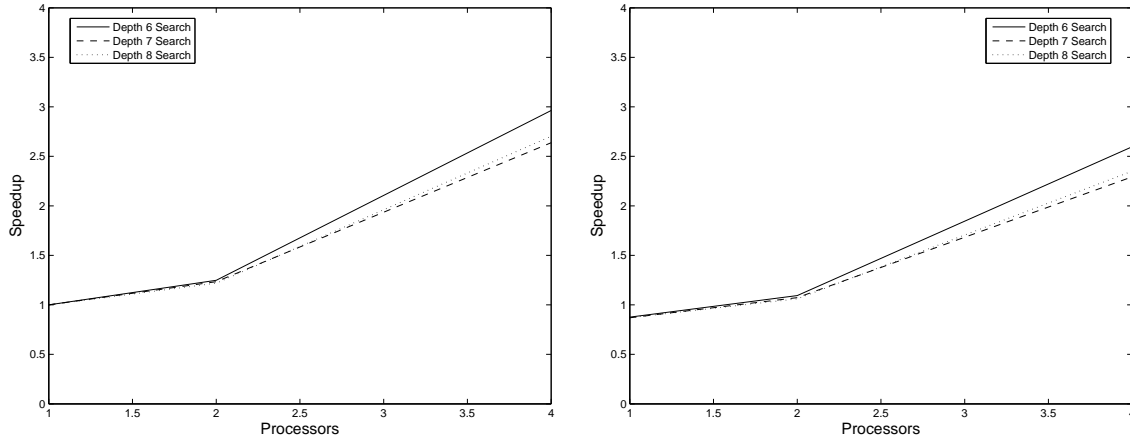
but as more processors are added, performance improves; close to 4 times speedup for six processors.

4.3.2 Search Tree Pattern

For the search-tree pattern, the experiment consists of running Games of Kece minimax searches to increasing depths. The results are obtained from a game with 31 words, played on a 30x30 kece board, where a player can place either of the top two remaining words on the board. For parallel executions, a depth 2 parallel expansion is performed before completing the rest of the search with sequential executions distributed among the processors. A depth 2 expansion for the game conditions used in testing creates around 40 nodes for distribution.

Figure 3(a) shows how a minimax search generated by the MAP₃S pattern performs when multiprocessor run times are compared against the MAP₃S code run on a single processor. Speedup of up to 3 times on four processors is observed. When compared to the sequential version of the Game of Kece, as depicted in Figure 3(b), the MAP₃S-generated code reduces in over 2.5 times speedup at four processors.

The pattern performs well for minimax searches; a significant speedup is observed for a four-processor execution of the pattern. This performance improvement likely also extends to optimization searches, but we do not expect to see similar improvements for alpha-beta searches. As currently implemented, the MAP₃S search-tree pattern does not allow for pruning to occur in the task-generating divide phase. For example, consider a parallel alpha-beta search whose divide phase is two-level deep. Running this search and a sequential alpha-beta search to the same depth would reveal that at the depth-2 the parallel search has more subtrees to search than the sequential search, simply because no pruning is possible for those first



(a) Speedup against a single processor parallel version of the Game of Kece

(b) Speedup against a sequential version of the Game of Kece

Figure 3: Speedup attained on the Game of Kece

two levels. Overall, this means that the parallel execution potentially could search far more alpha-beta tree nodes than the sequential search and, as such, will not produce significant performance improvements.

4.4 Comparison to $\text{CO}_2\text{P}_3\text{S}$

Table 1: Comparison between $\text{CO}_2\text{P}_3\text{S}$ and MAP_3S run times for the Game of Life (a mesh pattern on a shared-memory machine)

Processors	$\text{CO}_2\text{P}_3\text{S}$		MAP_3S		Ratio
	Runtime	Speedup	Runtime	Speedup	
Sequential	644.569	1.000	336.155	1.000	1.917
1	1020.040	0.632	538.871	0.624	1.893
2	543.896	1.185	246.223	1.365	2.209
3	356.472	1.808	170.940	1.963	2.082
4	268.150	2.404	135.768	2.476	1.975

Table 1 compares run times for implementations of the Game of Life on $\text{CO}_2\text{P}_3\text{S}$ and MAP_3S on a shared-memory four-processor SGI Origin 2100 machine. The Game is run for 4000 generations on a 600x600 board. Table 1 presents the runtime, in seconds, for each version of the Game of Life. Speedup is computed in relation to the corresponding sequential version of the program. The Ratio column shows the

speedup of MAP₃S over an equivalent execution of CO₂P₃S .

When the speedup of CO₂P₃S and MAP₃S in relation to the respective Java and C sequential versions of the code are compared, both systems produce very similar speedup. However, the sequential C version is almost twice as fast as the Java sequential version. Thus the MAP₃S version is consistently around twice as fast as the CO₂P₃S version. Another way of analyzing the data is to point out that the execution time with two processors in MAP₃S is shorter than with four processors in CO₂P₃S. The hope for MAP₃S is that HPC programmer’s familiarity with MPI and its better performance will facilitate its acceptance in the HPC community.

5 Related Work

This work builds on the experience and knowledge acquired with the development of CO₂P₃S [1, 7, 11]. An extensive review of related work is presented in MacDonald’s thesis [7].

Cole is developing the *eSkel* library that is also based on C and MPI [3, 5]. MAP₃S delivers in some of the principles enumerated by Cole, such as minimizing disruption in existing HPC programming infrastructure, allowing the integration of ad-hoc parallelism, and focusing on performance.

Dynamic load balancing by work stealing has been implemented in several multi-threaded systems, including Cilk [4] and EARTH [10]. The double-ended distributed-queue system used in the search-tree pattern appears to have been first described by Maquelin [9]. The beggar’s model for work stealing starting with randomly selected nodes was used in EARTH [6].

6 Conclusion

This paper described an initial effort toward building a generative design-pattern system that generates C code that uses MPI for communication and synchronization. While caution demands that the answer to the question in the title be withheld pending further investigation, the initial results are encouraging.

Future work on MAP₃S includes investigating further parameterization of existing patterns, implementing a larger suite of applications, and implementing other patterns available in CO₂P₃S and other pattern-based systems. The focus of this research is beyond the simple recreation of a CO₂P₃S system for MPI/C. The goal is to determine a set of performance parameters that can be provided to enable the tuning of parallelism concerns for alternate underlying architectures. The determination of these parameters and the development of tools to assist with performance tuning will require a fairly complete pattern generating system.

Acknowledgments

This work would not be possible without the extensive work of Steve MacDonald, John Anvik, Kai Tan, Jonathan Schaefer, and others in the development of CO₂P₃S. Special thanks to John Anvik for answering our questions, to Kai Tan for the assistance with distributed CO₂P₃S, and to Paul Lu for fruitful discussions. This research is supported by a Faculty Award from IBM Canada Ltd., by the Canadian Foundation for Innovation, and by grants from the Natural Science and Engineering Research Council of Canada (NSERC).

References

- [1] J. Anvik. Asserting the utility of CO₂P₃S using the Cowichan problems. Master's thesis, University of Alberta, Edmonton, AB, Canada, Dept. of Computing Science 2002.
- [2] J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Why not use a pattern-based parallel programming system? In *EuroPar International Conference on Parallel and Distributed Computing*, pages 48–57, Klagenfurt, Austria, 2003.
- [3] A. Benoit and M. Cole. Two fundamental concepts in skeletal parallel programming. In *Practical Aspects of High-level Parallel Programming*, Atlanta, GA, USA, May 2005.
- [4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing (JPDC)*, 37:55–69, August 1996.
- [5] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.
- [6] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.
- [7] S. MacDonald. *From patterns to frameworks to parallel programs*. PhD thesis, University of Alberta, Edmonton, AB, Canada, 2002. Dept. of Computing Science.
- [8] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.
- [9] O. Maquelin. *The ADAM architecture and its simulation*. PhD thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zurich, Switzerland, 1994.

- [10] J. N. Amaral P. Kakulavarapu, O. C. Maquelin and G. R. Gao. Dynamic load balancers for a multi-threaded multiprocessor system. *Parallel Processing Letters*, 11(1):169–184, March 2001.
- [11] K. Tan. Pattern-based parallel programming in a distributed memory environment. Master’s thesis, University of Alberta, Edmonton, AB, Canada, 2003. Dept. of Computing Science.
- [12] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 203–215, San Diego, CA, U.S.A., 2003.
- [13] G. Wilson. Assessing the usability of parallel programming systems: The cowichan problems. In Karsten Dekker and Rene Rehmman, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 183–193. Birkhauser Velag, Basel, Switzerland, 1994.