A Case Study of Shared Memory and Message Passing: The Triangle Puzzle

Kevin Lew* and Kirk Johnson

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139
{lew, tuna}@lcs.mit.edu

1 Introduction

We completed a case study [4] that compares shared-memory and message-passing implementations of an algorithm that counts the solutions to the triangle puzzle. Previous relevant case studies [2, 5] have resulted to simulation, whereas our implementations run on a 32-node MIT Alewife machine, a cache-coherent, distributed shared-memory multiprocessor that efficiently supports both the shared-memory and message-passing programming models [1]. Only the communication interfaces used by the implementations vary; all other system components remain fixed.

The triangle puzzle consists of a small wooden triangular board, with fifteen holes in the shape of an equilateral triangle. As shown in Figure 1, a player begins with fourteen pegs and a hole in the middle of the third row. The object of the puzzle is to make a sequence of moves after which one peg remains. Each move consists of jumping over one peg with another and removing the peg, as in checkers. Counting the total number of sequences of moves leading to a solution is the goal of the *triangle puzzle search problem*. *Problem size* refers to the number of holes per side in the puzzle board. A *position* is a particular placement of pegs into holes. An *extension* is a position that can be derived from a given position by making one move.

We chose the triangle puzzle because not only is it simple to understand and explain, but it also exhibits some of the characteristics of more complex tree-search problems, such as chess [3]. In addition, the parallel applications that solve the search problem are fine-grained and irregular. They are *fine-grained* because the time between communication events is short. A *communication event* consists of either a shared-memory reference or a user-level message between two processors. The parallel applications are *irregular* because the data structures used evolve dynamically as the applications execute.

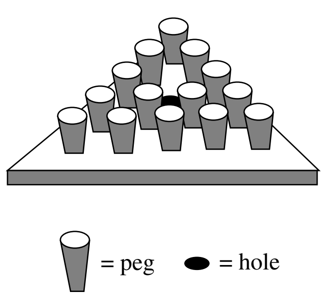

Figure 1. Initial position for problem size 5.

Section 2 explains the shared-memory and message-passing implementations that solve this search problem. Finally, Section 3 concludes.

2 Parallel Solutions to the Triangle Puzzle

To solve the triangle puzzle search problem, the shared-memory and message-passing implementations construct an exhaustive search tree in a parallel breadth-first manner. The root of this search tree represents the initial position, and all proper descendants represent positions that can be derived from the initial position by making one or more moves. To reduce the size of the search tree, three axes of reflective symmetry of the puzzle board are exploited, as shown in Figure 2. By performing one or more reflections about these axes, more than one position can be stored per node in the search tree, thus making the tree smaller than if symmetry were not exploited.

In addition to exploiting symmetry, a transposition table is used to store all positions that the search procedure has already seen. This table is implemented as a large hash table. Thus, if a position is found a second time, the extensions derived from this position will not be generated a second time. Because the same position can often be reached via two different sequences of moves, the search tree is in fact a directed acyclic graph (dag).

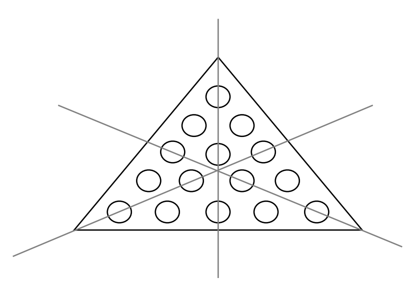

Figure 2. Three axes of reflective symmetry for problem size 5.

*This research was performed under the supervision of Professor M. Frans Kaashoek, and was supported in part by ARPA Contract N-00014-94-1-0985 and in part by an NSF National Young Investigator Award.

To count the total number of solutions, each node in the search dag has a solution counter, whose value is the number of paths from the root to this node. The root's solution counter is set to 1, then this value trickles down to all proper descendants, such that the value of a solution counter of one node equals the sum of the counter values of its parents. To obtain the total number of solutions, after the entire search dag is completed, the counter values of all leaves that represent positions with only one peg remaining are summed.

This search dag is constructed in a parallel breadth-first fashion. Before the first level of the search dag is constructed, the initial position is placed in a *current* queue, which is a list of positions to extend in the current level of the search dag. Then, each level of the search dag is constructed in the following manner. Processors, in parallel, fetch positions to extend from the *current* queue. Each extension is then looked up in the transposition table. If an extension is not in the table, it is added to the table and to another queue, called the *next* queue, which is a list of positions to extend in the next level of the search dag. (The *current* and *next* queues are collectively known as *work queues*.) If an extension is already in the transposition table, the value of the solution counter of the position in the table is increased by that of the extension. When the *current* queue is empty, a level of the search dag is completed. The positions in the *next* queue are placed in the *current* queue, the transposition table is cleared, and then the next level is constructed.

One shared-memory implementation partitions the transposition table across all processors, and stores a private set of work queues in each processor's shared-memory bank. To distribute work, the bit-vector representations of extensions are hashed to determine a destination processor that receives the extension. Extensions are copied via shared-memory loads and stores.

A message-passing implementation performs exactly the same search algorithm as this shared-memory implementation, and distributes data structures in a similar manner, except the data structures are placed in the *private* memory banks of each processor. Instead of distributing extensions via shared-memory loads and stores, this message-passing implementation distributes extensions by sending user-level messages between processors. Again, the receiving processor is determined by hashing the bit representation of an extension.

Thus, these shared-memory and message-passing implementations perform exactly the same search algorithm, except they differ in the communication mechanism used. To distribute extensions, the shared-memory version uses memory loads and stores, which cause cache coherence actions and decouple locking (synchronization) and data transfer. The cache coherence actions and locking are implemented with *multiple* round-trip messages. In contrast, the message-passing version obtains the same functionality with a *single* user-level message. Thus, the shared-memory implementation is less efficient than the message-passing implementation. In fact, the message-passing implementation performs up to 52% better than this shared-memory implementation for problem size 6.

Another shared-memory implementation eliminates the overhead of hashing in distributing extensions by placing all data structures in a single processor's shared-memory bank. By running a smaller problem size (size 5), we observe that this shared-memory implementation performs better than the previous shared-memory implementation and the message-passing implementation. In fact, this shared-memory implementation performs up to 14% better than the message-passing implementation. The principal reason is that by reducing the problem size, contention for the work queues is lower.

3 Conclusions

There are other shared-memory implementations [4], but they reinforce the two main conclusions of this case study. First, when using distributed cache-coherent shared memory, performing cache coherence actions and decoupling synchronization and data transfer can make shared memory less efficient than message passing. Second, shared memory can perform better than message passing for applications that exhibit low contention for data structures.

Thus, depending on what one is considering, shared memory can perform better than message passing, or vice versa. Therefore, we advocate considering using both communication mechanisms when designing parallel applications.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. "The MIT Alewife Machine: Architecture and Performance." To appear in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[2] S. Chandra, J. Larus, A. Rogers. "Where Is Time Spent in Message-Passing and Shared-Memory Implementations?" In *Proceedings of Architectural Support for Programming Languages and Operating Systems VI*, San Jose, California, October 1994, pp. 61-73.

[3] B. Kuszmaul. *Synchronized MIMD Computing*. Ph.D. thesis. Massachusetts Institute of Technology. May 1994.

[4] K. Lew. *A Case Study of Shared Memory and Message Passing: The Triangle Puzzle*. Master's thesis. Massachusetts Institute of Technology. January 1995.

[5] M. Martonosi and A. Gupta. "Trade-offs in Message Passing and Shared Memory Implementations of a Standard Cell Router." In *Proceedings of the 1989 International Conference on Parallel Processing*, Pennsylvania State University Park, Pennsylvania, August 1989, pp. III-88 to III-96.