A Case Study of Shared Memory and Message Passing:

The Triangle Puzzle

by

Kevin A. Lew

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 20, 1995

Author ....
Department of Electrical Engineering and Computer Science
January 20, 1995

Certified by ....
Professor M. Frans Kaashoek
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ....
Kirk Johnson
Department of Electrical Engineering and Computer Science
Thesis Co-Supervisor

Accepted by ....
Professor Frederic R. Morgenthaler
Chair, Department Committee on Graduate Students
Department of Electrical Engineering and Computer Science

A Case Study of Shared Memory and Message Passing:

The Triangle Puzzle

by

Kevin A. Lew

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 1995, in partial fulfillment of the requirements for
the degree of Master of Science

Abstract

This thesis is the first controlled case study that compares shared-memory and message-passing implementations of an application that solves the triangle puzzle and runs on actual hardware: only the communication interfaces used by the implementations vary; all other system components remained fixed. The implementations run on the MIT Alewife machine, a cache-coherent, distributed-shared-memory multiprocessor that efficiently supports both the shared-memory and message-passing programming models. The goal of the triangle puzzle is to count the number of solutions to a simple puzzle in which a set of pegs, arranged in a triangle, is reduced to one peg by jumping over and removing a peg with another, as in checkers. The shared-memory implementations explore distributing data structures across processors' memory banks, load distribution, and prefetching. A single message-passing implementation uses only message passing for interprocessor communication. By comparing these shared-memory and message-passing implementations, we draw two main conclusions. First, when using distributed shared memory, performing cache coherence actions and decoupling synchronization and data transfer can make a shared-memory implementation less efficient than the message-passing implementation. For our application, we observe a maximum of 52% performance improvement of the message-passing implementation over the best shared-memory one that uses both synchronization and data transfer. Second, shared memory offers low-overhead data access and can perform better than message passing for applications that exhibit low contention. For our application, we observe a maximum of 14% improvement of a shared-memory implementation over the message-passing one. Thus, sometimes message passing is better than shared memory. Sometimes the reverse is true. To enable all parallel shared-memory and message-passing applications to perform well, we advocate parallel machines that efficiently support both communication styles.

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

Thesis Co-Supervisor: Kirk Johnson
Title: Doctoral candidate

Acknowledgments

I wish to thank several people who made this thesis possible. First and foremost, I thank Prof. M. Frans Kaashoek for introducing me to the problem presented in this thesis and for caring by holding weekly meetings to stimulate my thinking and provide guidance. Second, I thank Kirk Johnson for providing me the code that solves the triangle puzzle on the CM-5, his guidance, and the numerous enlightening discussions as this work evolved. Next, I wish to thank the members of my research group for providing feedback and an excellent learning environment: Dawson Engler, Sandeep Gupta, Wilson Hsieh, Max Poletto, Josh Tauber, and Deborah Wallach. I would also like to thank the members of the Alewife group for providing a truly unique machine, especially Rajeev Barua, David Chaiken, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, Sramana Mitra, and Donald Yeung. Thanks also to Kavita Bala, Chris Lefelhocz, and Ken Mackenzie for providing useful feedback on earlier drafts of this thesis. Finally, thanks to my family for their continuous encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advocates of shared-memory multiprocessors have long debated with supporters of message-passing multicomputers. The shared-memory camp argues that shared-memory machines are easy to program; the message-passing camp argues that the cost of hardware to support shared memory is high and limits scalability and performance. This thesis is the first controlled case study of shared-memory and message-passing implementations of an application that solves the triangle puzzle and runs on actual hardware: only the communication interfaces used by the implementations vary; all other system components (e.g., compiler, processor, cache, interconnection network) remain fixed. We show when and why the message-passing version outperforms a shared-memory version and vice versa. Additional overhead due to cache coherence actions and decoupling synchronization and data transfer can make a shared-memory implementation perform worse than the message-passing implementation (our message-passing implementation performs up to 52% better than the best shared-memory implementation that uses both synchronization and data transfer). A shared-memory version can outperform the message-passing implementation (by up to 14% for our application) under low contention because shared memory offers low-overhead data access.

Our implementations run on the MIT Alewife multiprocessor [2]. The message-passing implementation was ported from a message-passing implementation that runs on Thinking Machines' CM-5 family of multicomputers [25]. The original CM-5 implementation written by Kirk Johnson won first place in an Internet newsgroup contest [14], the goal of which was to solve the triangle puzzle in the shortest time.

Alewife efficiently supports both message-passing and cache-coherent shared-memory programming models in hardware. In fact, Alewife is the only existing machine of its class, and is thus a unique platform on which to compare shared-memory and message-passing implementations. Previous research comparing shared-memory and message-passing implementations of the same application has resorted to either using the same machine to run different simulators [8] or using different machines to run different simulators [19]. The Stanford FLASH multiprocessor [17] also efficiently supports these two programming models, but has yet to be built.

We chose the triangle puzzle because it is simple enough to understand and solve, yet exhibits many of the characteristics of complex tree-search problems. For example,

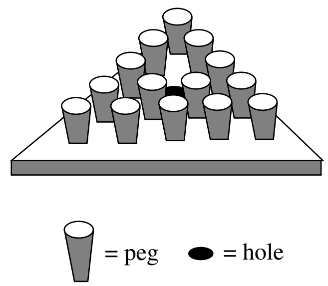

Figure 1.1. Initial placement of pegs in the triangle puzzle.

exploiting the symmetry of the playing board is used both in our implementations and in game-search problems, such as chess. In addition, solving this problem involves many parallel programming issues, such as load distribution, data management, and synchronization. Finally, our application is *fine-grained* and *irregular.* By "fine-grained," we mean that the time spent computing between shared-memory accesses or communication events is short. Although our study is performed using a single fine-grained application, we expect that our conclusions will hold for other fine-grained applications. Our application is irregular because its data structures evolve dynamically as the application executes.

The triangle puzzle consists of a triangular board with fifteen holes, each of which may contain a peg. At the start of the puzzle, every hole except the middle hole of the third row contains a peg (see Figure 1.1). A move consists of removing a peg by jumping over it with another peg as in checkers. A solution to the puzzle is a sequence of moves that leaves only one peg on the board. Counting the number of distinct solutions to the triangle puzzle is the goal of the *triangle puzzle search problem* [15]. Because we must find all solutions, solving this search problem involves an exhaustive search. This search problem can be extended for puzzles with boards that have more than five holes on each side. We refer to the number of holes per side as the *problem size*.

We solved problem sizes 5, 6, and 7 on a 32-node Alewife machine. Problem size 5 has 1,550 solutions. Problem size 6 has 29,235,690,234 solutions. Problem size 7 has *zero* solutions. Gittinger, Chikayama, and Kumon [13] show that there are zero solutions for problem sizes $3N+1$ ($N \geq 2$). Solving problem size 8 is expected to require several gigabytes of memory [14]; to the best of our knowledge, no one has solved it yet.

The rest of this thesis is organized as follows. Chapter 2 provides background on the architectural, programming, and performance issues when using shared-memory and message-passing machines. Chapter 3 describes how we solve the triangle puzzle using parallel tree search. Chapter 4 describes and compares the performance of our shared-memory and message-passing implementations, describes an implementation that uses both shared memory and message passing, and closes by presenting the speedups of the message-passing implementation and the best shared-memory implementation. Chapter 5 presents related work. Finally, Chapter 6 concludes.

Chapter 2

Background

This chapter presents background material on the architectural, programming, and performance issues of shared-memory and message-passing machines.

2.1 Architectural Issues

Shared memory and message passing are two interprocessor communication mechanisms. Multiple processors can share memory (i.e., use a single address space) in two ways. First, all processors can access a single memory bank by sending messages across a network, as shown in Figure 2.1. This organization is not scalable because the single shared-memory bank becomes a bottleneck when the number of processors becomes large. Second, all processors can access multiple memory banks in a "dance hall" fashion, as shown in Figure 2.2. Machines with this "dance hall" organization solve the bottleneck problem, but can be improved by tightly coupling each processor with a memory bank, as shown in Figure 2.3. Machines with this tightly coupled processor-memory organization are called distributed-shared-memory machines, and are better than those with the "dance hall" organization because the wires between processors and memory banks are shorter. Thus, distributed-shared-memory machines cost less, and a given processor takes less time to access its tightly coupled memory bank.

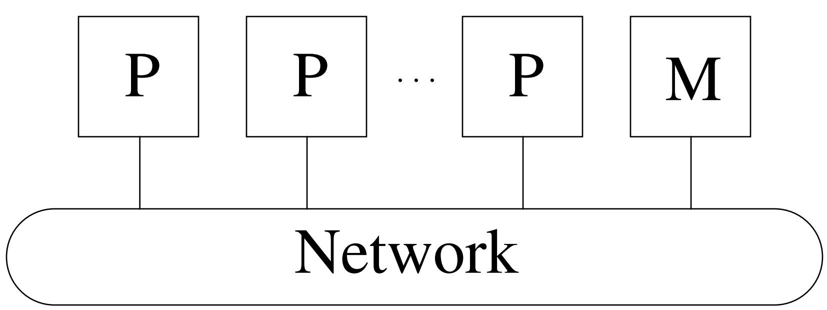

Figure 2.1. Shared-memory machine with one memory bank.
P = processor, M = memory bank.

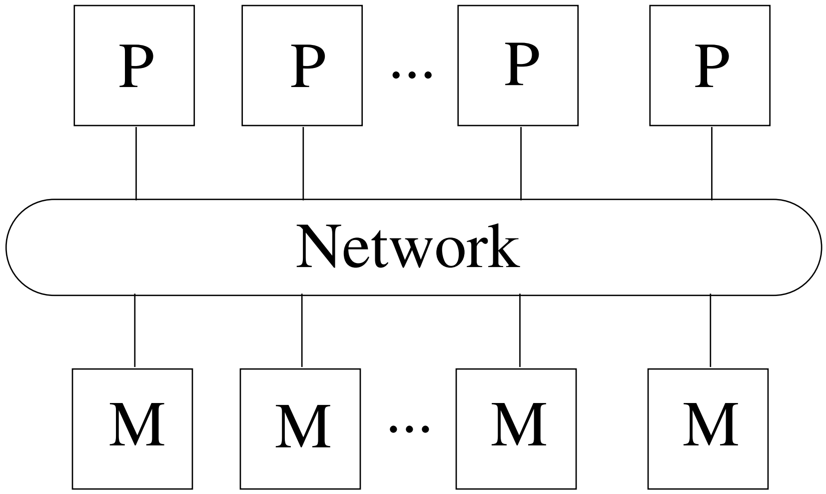

Figure 2.2. Machine with "dance hall" organization.
P = processor, M = memory bank.

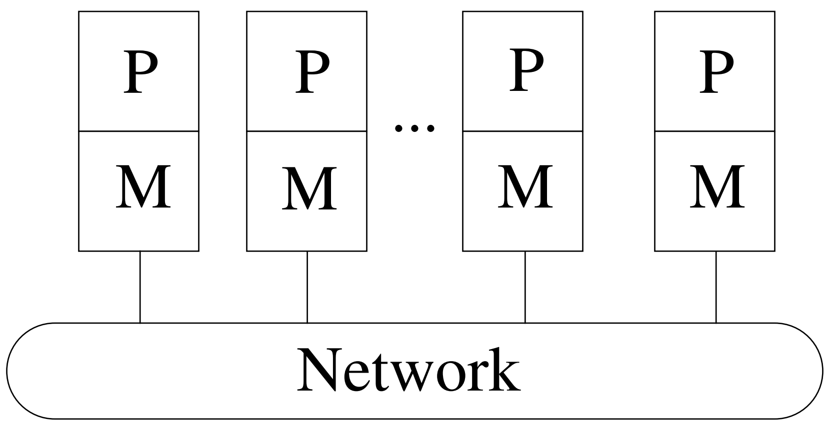

Figure 2.3. Machine with processors tightly coupled with memory banks.
P = processor, M = memory bank.

Caching also helps reduce memory access time. If the tightly coupled processor-memory organization is used to implement shared memory, and each processor has a cache, then *every* memory reference to address *A* must perform the following check: if the data with address *A* is currently cached, then load it from the cache. Otherwise, if *A* is a local address, then load the data from the current processor's memory bank. If none of these conditions are true, then load the data from remote memory and perform cache coherence actions. To minimize the time taken by this procedure of checks, it is implemented with specialized hardware. By using this hardware support, shared-memory accesses can be performed using a single instruction [16].

To illustrate cache coherence actions, we describe actions that a directory-based cache coherence protocol executes when a processor references remote data that is not present in its cache. In a directory-based cache coherence protocol, a section of memory, called the *directory*, stores the locations and state of cached copies of each data block. To invalidate cached copies, the memory system sends messages to each cache that has a copy of the data. In order to perform a remote read, a processor must send a request to the remote processor, which then responds with the requested data. In addition, the data must be placed in the requesting processor's cache, and the directory of cached copies in the remote processor must be updated. In order to perform a remote write, a processor sends a request to the remote processor, which then must invalidate all cached copies of the data.

Some message-passing machines organize processors, memory banks, and the network as shown in Figure 2.3. However, instead of having a single address space for all processors, each processor has its own address space for its local memory bank. To access a remote processor's memory bank, a processor must explicitly request data by sending a message to the remote processor, which then responds with a message containing the requested data. (Of course, a shared-memory programming model (i.e., a single address space) can be implemented using this message-passing architecture; messages are used to keep data values consistent across all processors [5, 6].) Thus, the check described in the previous paragraph is not performed by hardware in message-passing machines. This is the main difference between shared-memory and message-passing machines. If this check is to be performed, it must be implemented entirely in software using multiple instructions.

2.2 Programming Issues

Because shared-memory machines offer programmers a single address space whereas message-passing machines offer multiple address spaces, shared-memory machines are conceptually easier to program. For a shared-memory machine with multiple distributed shared-memory banks, the shared-memory programming model makes programming easier because it frees the programmer from having to manage multiple memory banks and it hides message passing from the programmer. If a programmer uses message passing, the programmer is in charge of orchestrating all communication events through explicit sends and receives. This task can be difficult when communication is complex.

For shared-memory machines, synchronization (e.g., locks and barriers) is usually provided by various functions in a library. Data transfer is accomplished by memory loads and stores. On the other hand, when using message passing, synchronization and data transfer can be coupled. When one processor transfers data to another processor by sending the data in a message, the receiving processor can process the message by atomically executing a block of code, as in active messages [26]. Because this processing is performed atomically, this code can be thought of as a critical section; it is if a lock were acquired, the code is executed, and the lock were released.

In the following section, we discuss the performance implications of this coupling of synchronization and data transfer in message passing versus their being decoupled in shared memory. We assume that all synchronization and data transfers are achieved by using simple library calls.

2.3 Performance Issues

There are certain scenarios in which shared memory can perform worse than message passing [16]. First, if the grain size of shared data is larger than a cache line, shared memory may perform worse because a data transfer requires multiple coherence actions, which demand more network bandwidth and increase the latency of the data transfer. It would be more efficient to send the data using a single message. Even if prefetching were used, shared memory can still perform worse, partly because issuing load instructions to perform prefetching incurs overhead, and the prefetched data consumes network bandwidth. Still, the main reason shared memory, even with prefetching, performs worse than message passing when the grain size of shared data is larger than a cache line is that the overhead of cache coherence actions increases the demand on network bandwidth and increases the latency of data transfer. In addition, if data is immediately consumed after it is obtained and not re-used, the time used performing cache coherence actions is not well spent; the main idea behind caching is to reduce latency by exploiting the fact that if data is accessed once, it is likely to be referenced again.

Second, shared memory can perform worse when communication patterns are known and regular. The main reason is that shared-memory accesses requires two messages: one message from the requesting processor to the shared-memory bank, and one from the memory bank back to the requesting processor. On the other hand, message passing is inherently one-way: a message is sent from one point to another. In addition, in many cache coherence protocols, for a processor to acquire a cache line that is dirty in another processor's cache, the data must be transferred through a home or intermediate node instead of being communicated directly to the requester. For example, in a protocol that keeps the status of a cached data block with a home node, the home node must update the status of the data block.

Lastly, shared memory can perform worse than message passing in combining synchronization and data transfer, assuming that sending a message requires very little overhead. As described in the previous section, message-passing machines can bundle synchronization with data transfer by using mechanisms such as active messages, whereas shared-memory machines decouple the two operations. By sending a one-way, point-to-point message, a message-passing machine can effectively acquire a lock, transfer data, and release the lock. On the other hand, in shared memory, this procedure requires the following steps. One round trip is required to obtain the lock: a processor sends a request message to the memory bank where the lock is located, the lock is obtained, and a response message is sent back to the requester. Then, data transfer takes place. Finally, one round trip, analogous to acquiring the lock, is required to release the lock. Because message passing bundles synchronization and data transfer in a one-way, point-to-point message, it can perform better than shared memory, which requires multiple round-trip messages to achieve the same functionality.

In summary, shared memory can perform worse than message passing under three conditions: (1) when the grain size of shared data is larger than a cache line, (2) when communication patterns are known and regular, and (3) when combining synchronization and data transfer. When grain size of shared data is small and communication patterns are irregular, shared memory can perform better than message passing.

Chapter 3

Solving the Triangle Puzzle

This chapter describes using tree search to solve the triangle puzzle, using a transposition table to eliminate redundant work during tree search, exploiting symmetry to optimize the search process, and using parallel breadth-first search.

3.1 Structure of the Solution

The generic solution to the triangle puzzle constructs a search tree. Each node in the search tree represents some placement of pegs into holes that can be derived from the initial placement of pegs through a sequence of moves. Any such placement is called a *position*. We represent a position as a bit vector, with each bit corresponding to whether a peg is present in a particular hole. An edge from node A to node B indicates that the position represented by B can be obtained by making a single move from the position represented by node A. We call the position represented by node B an *extension* of the position represented by node A. Thus, the root of the search tree represents the initial position, and leaves of the tree represent positions from which no more moves can be made. For an initial position with P pegs, a sequence of P-1 moves is required to arrive at a solution to the triangle puzzle, since each move removes exactly one peg. Therefore, a path from the root to any leaf at a depth of P-1 in the search tree is a solution. We call such leaves *valid leaves*. All other leaves at depths less than P-1 are *invalid leaves*.

Positions are stored in a *transposition table* so that extensions of positions that have already been explored are not explored again. When an extension of a position is found in the transposition table, the subtree generated from this extension does not need to be explored again, and we can instead *join* the extension and the position in the transposition table (this is also known as *folding* or *fusing* positions) [1, 23]. In the triangle puzzle, because the number of joined positions is large, the size of the search tree is greatly reduced by using a transposition table. For problem sizes 5, 6, and 7, 66%, 83%, and 90% of all positions explored are joined, respectively. Because joining occurs, what we refer to as the search tree is actually a directed acyclic graph (dag).

The transposition table is implemented using a hash table. Similar to an element of an array, a *slot* of the transposition table holds a set of positions that map to that slot under a

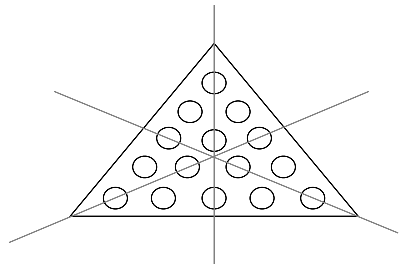

Figure 3.1. Axes of symmetry of the triangle puzzle board.

hash function. If two positions map to the same slot, the positions are chained together in the slot.

To further reduce the number of positions explored, our implementations also exploit the symmetry of the triangle puzzle board. Since the board is an equilateral triangle, there are three axes of symmetry, as shown in Figure 3.1. Thus, up to six positions can be represented in each node of the search dag by performing the appropriate reflections. If only one reflection about one axis of symmetry is performed, each node in the search dag represents two positions (not necessarily distinct). A node can be uniquely identified by choosing one of the positions as the *canonical position*. (We interchangeably refer to a node and a canonical position as a position.) Since a position in our triangle puzzle implementations is represented as a bit vector, we arbitrarily choose the position that lexicographically precedes the other as the canonical position. For problem sizes 5, 6, and 7, if each node represents two positions, the number of positions explored is nearly halved, as shown in Table 3.1.

If each node in the search dag represents six positions, all six positions can be obtained by making a sequence of five reflections about two axes of symmetry, by alternating between the axes on each reflection in the sequence. For problem sizes 5 and 6, if each node represents six positions, the number of positions explored is reduced by nearly a factor of two (exactly the same factor as when each node represents two positions), as shown in Table 3.1. However, doing the same for problem size 7 cuts the number of positions explored by a factor of almost six [4].

Clearly, there is a time-space trade-off when exploiting symmetry: if the search procedure exploits symmetry, extra computation is required to perform the reflections, but the search procedure saves space and computation by reducing the size of the search dag and avoiding computing positions already explored, respectively.

Problem Size	Symmetry (number of positions per node)	Number of Positions Explored	Reduction Factor Compared to 1 Position Per Node
5	1	4,847	1.000
	2	2,463	1.968
	6	2,463	1.968
6	1	1,373,269	1.000
	2	688,349	1.995
	6	688,349	1.995
7	1	304,262,129	1.000
	2	152,182,277	1.999
	6	53,158,132	5.724

Table 3.1. Relationship between symmetry and number of positions explored.

In addition to using a transposition table and exploiting symmetry, other algorithmic optimizations may be performed during tree search. These include compression [1], evicting positions from the transposition table when a threshold is reached [4, 18], pre-loading the transposition table [1], pruning heuristics besides exploiting symmetry [10, 21], and move ordering [18]. We did not explore the first two optimizations, and we do not employ the remaining ones. We cannot pre-load the transposition table because information obtained at one point in the search gives no information about later stages of the search. Finally, because the triangle puzzle search problem requires finding all solutions and hence requires an exhaustive search, we do not use any pruning heuristics other than exploiting symmetry, and we do not perform move ordering. Move ordering helps when different moves have different effects on the outcome (e.g., a score) of a puzzle or game. In the triangle puzzle, all sequences of moves are equal in value.

3.2 Parallel Breadth-First Search versus Parallel Depth-First Search

Parallel breadth-first search (PBFS) executes faster and uses less memory than parallel depth-first search (PDFS) when used to solve the triangle puzzle. Intuitively, PBFS is more appropriate for the triangle puzzle than PDFS because PBFS inherently performs an exhaustive search to find all solutions. On the other hand, if we had a search problem whose objective was to find any solution, PDFS would probably be more suitable. The performance difference is a result of how the total number of solutions is determined. We keep a *solution counter* with each node that stores the number of paths from the root that can reach this node. PBFS sets the counter in the root of the search dag to “1,” then each node at the next level sums the values of the counters of its parents. The sum of the values of the counters of valid leaves is the number of solutions to the triangle puzzle.

In contrast, in PDFS, since different processors can be working concurrently on different levels in the dag, PDFS cannot count solutions in the same top-down manner as in PBFS. The search procedure has two options: it can either “push” counter values down complete subdags in a manner similar to counting solutions in PBFS, or it can count solutions in a bottom-up fashion after the entire dag is formed. With the first option, the search procedure can traverse portions of the dag multiple times. For example, suppose one processor is exploring a subtree. Now suppose that another processor generates an extension that is represented by the root of this subtree. Then the search procedure must “push” the counter value of this extension down the entire subtree in order to update the number of paths that can reach all nodes in the subtree. This process is very costly because of the high number of joins, as noted in Chapter 3.1.

To count solutions in a bottom-up fashion in PDFS, the search procedure sets the counters of valid leaves to “1,” and the counters of invalid leaves to “0.” Starting at the bottom of the search dag and working upwards, nodes at each level of the dag then sum the values of their children’s counters. The value of the counter at the root of the dag is the number of solutions. Because PDFS that counts solutions in a top-down fashion can traverse portions of the dag multiple times, whereas PDFS that counts solutions in a bottom-up fashion traverses the dag only twice, the latter performs better. Henceforth, “PDFS” refers to the latter version.

PBFS executes faster than PDFS because PBFS traverses the search dag once, whereas PDFS traverses it twice: once to construct the dag, and a second time to count solutions. In addition, detecting when the dag is completely formed in PDFS incurs additional overhead.

PBFS also requires less memory than PDFS. Since PBFS counts solutions in a top-down fashion, it needs to remember at most only those positions in two successive levels of the dag[1]. On the other hand, because PDFS counts solutions in a bottom-up fashion, it needs to remember all positions in the dag. Thus, PBFS will require less memory than PDFS to store positions in the transposition table, and it will require progressively less

1. This is a loose upper bound because once all extensions have been generated from a given position, the position can be discarded.

Problem Size	Total Number of Positions Explored	Largest Number of Positions at Two Successive Levels of the Search Dag
5	2,463	356
6	688,349	44,219
7	53,158,132	1,815,907

Table 3.2. Position statistics.

memory than PDFS as the problem size increases. This point is illustrated in Table 3.2, which shows the total number of positions explored and the largest number of positions at two successive levels for problem sizes 5, 6, and 7, when symmetry is exploited.

We developed an implementation of PDFS that counts solutions in a bottom-up fashion and runs on Thinking Machines' CM-5 family of multicomputers. The performance of this PDFS implementation for problem size 5 and 64 processors (.046 seconds, averaged over ten runs) is a factor of five worse than that of the CM-5 PBFS implementation (.009 seconds, averaged over ten runs), thus supporting the argument that PBFS is better than PDFS for the triangle puzzle. We expect that this is also true for larger problem sizes because the size of the search dag significantly increases as problem size increases. Therefore, we only implemented PBFS on Alewife.

3.3 Implementation of PBFS

PBFS uses four data structures: a transposition table, a *current* queue of positions to extend at the current level in the search dag, a *next* queue of positions to extend at the next level, and a pool of positions (see Figure 3.2). We refer to the *current* and *next* queues collectively as *work queues*. The pool of positions is a chunk of memory used to store positions; to keep the counters associated with positions consistent, the transposition table and the work queues use pointers to refer to positions in this pool.

PBFS manipulates these data structures in a number of rounds equal to the number of levels in a complete search dag. Before the first round begins, the initial position is placed in the *current* queue. Each round then proceeds as follows. Processors repeatedly remove positions from the *current* queue and generate all possible extensions of each position. If an extension is not already in the transposition table, it is placed in both the transposition table and the *next* queue, with the solution counter set to the value of the counter of the

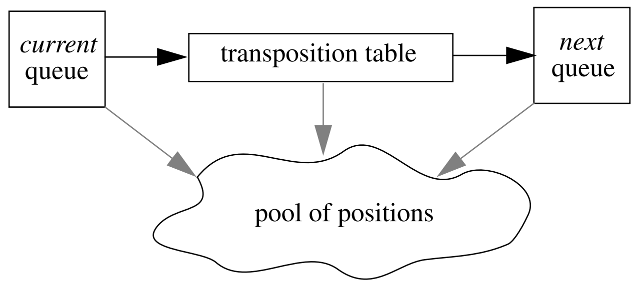

Figure 3.2. Four data structures used in PBFS. Solid arrows indicate the path travelled by extensions. Dashed arrows indicate that the *current* queue, *next* queue, and the transposition table use pointers to refer to positions in the pool of positions.

position from which this extension was generated. If an extension is already in the transposition table, it is joined with the position in the table by increasing the solution counter of the position in the table by the value of the counter of the extension, and the extension is not placed in the *next* queue. After all positions in the *current* queue have been extended, the round ends. Positions in the *next* queue are placed in the *current* queue, and the transposition table is cleared. Then the next round starts. After the last round is completed, the sum of the values of the solution counters associated with positions in the *current* queue is the number of solutions.

Chapter 4

Implementations and Performance Results

This chapter presents hardware details of Alewife, describes our shared-memory and message-passing implementations, and compares their performance. The shared-memory implementations explore distributing the transposition table and privatizing the work queues, distributing load, and prefetching. After comparing the shared-memory implementations with a message-passing one, we show that a hybrid shared-memory and message-passing implementation is not suitable for this application. We close by comparing the speedups of the message-passing implementation and the best shared-memory implementation.

Most of the results we present are for problem size 6. The reason we show results mostly for problem size 6 is that this problem size is large enough to produce significant changes in execution time as we vary the number of processors, yet not so large that it takes an excessive amount of time to solve. For example, the best parallel implementation (our message-passing implementation) takes about 200 times longer to solve problem size 7 than to solve problem size 6 on 32 processors. In addition, at least 32 processors are needed to solve problem size 7 because of memory requirements. Thus, for problem size 7, it would be impossible to draw conclusions about how execution time changes as the number of processors varies since the present maximum machine size is 32 processors.

The execution time of each implementation is averaged over ten trials. The standard deviation for all experiments is less than .03 seconds (worst case), and is usually much less (.0028 seconds average for all problem size 6 experiments and .000076 seconds average for problem size 5). The execution time of each trial is measured by using a cycle counter. Because setup times vary among different implementations, we do not include setup time in our measurements, unless otherwise noted. To supplement the graphical presentation of the performance of each implementation, Appendix A numerically presents the execution times and standard deviations of all implementations. Appendix A also shows the number of positions explored and the number of joined positions per round for problem sizes 5, 6, and 7.

As a final note, the total memory space required by the transposition table, summed over all processors, is held constant for all shared-memory and message-passing implementations for a given problem size. (i.e., the number of slots and the maximum number of positions a slot holds are constant for a given problem size.) In addition, the pool of

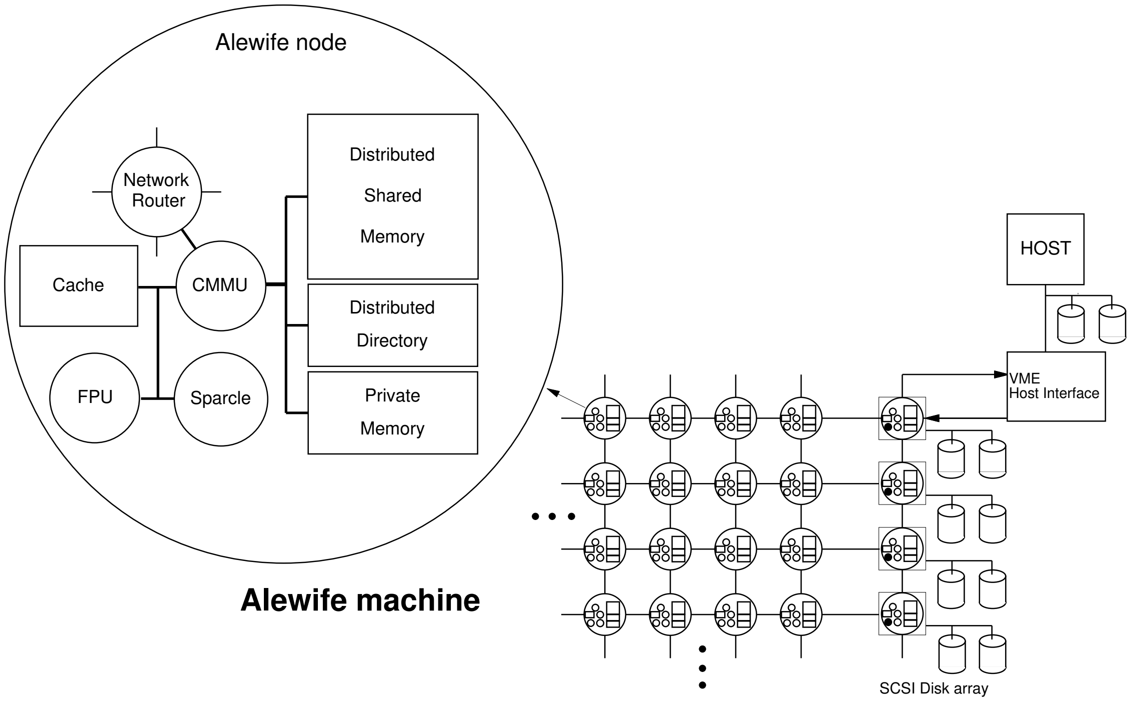

Figure 4.1. The Alewife architecture.

positions is equally partitioned among all processors for all shared-memory and message-passing implementations.

4.1 Alewife Hardware Details

The results presented in the following sections were obtained on a 32-node Alewife machine at the MIT Laboratory for Computer Science. Alewife is a distributed shared-memory, cache-coherent multiprocessor. Each processor is tightly coupled with a memory bank. Each node contains its own memory bank, part of which is used as a portion of the single shared address space. Each node consists of a Sparcle processor [3] clocked at 20MHz, a 64KB direct-mapped cache with 16-byte cache lines, a communications and memory management unit (CMMU), a floating-point coprocessor, an Elko-series mesh routing chip (EMRC) from Caltech, and 8MB of memory [2]. The EMRCs implement a direct network [24] with a two-dimensional mesh topology using wormhole routing [11]. A mesh-connected SCSI disk array provides I/O. Figure 4.1 shows the Alewife architecture. The CMMU implements shared-memory and message-passing communication interfaces, which will be described next.

4.1.1 Shared-Memory Interface

Shared memory is accessed by loads and stores. Caches store recently accessed shared data. These caches are kept coherent by a directory-based protocol called LimitLESS [7]. Directory-based protocols were described in Chapter 2.1. LimitLESS implements the directory in both hardware and software. Each processor keeps six hardware pointers to cached data blocks: one pointer consists of a single bit to indicate whether a word is cached at the local processor and five 32-bit pointers refer to remote processors' caches. When more pointers are required, a processor is interrupted for software emulation of a larger directory. We refer to this emulation as "time spent executing LimitLESS software." Because there are six hardware pointers, no time is spent executing LimitLESS software with six and fewer processors. Of the 8MB of memory per node, 4MB are used as a piece of global shared memory, 2MB are used for cache coherence directories, 1MB is user-accessible private memory, and 1MB is used by the operating system. A clean read miss to a processor's local shared memory bank is satisfied in about 13 cycles, while a clean read miss to a neighboring processor's memory bank takes about 38 cycles. Cached 32-bit shared-memory loads and stores require two and three cycles, respectively.

4.1.2 Message-Passing Interface

Besides providing a shared-memory interface, Alewife provides a message-passing interface, which allows programmers to bypass shared-memory hardware and access the communication network directly. This organization is illustrated in Figure 4.2. When a processor sends a message to another processor, the receiving processor is interrupted and then processes the message. This message-passing model provides the functionality of active messages [26]. The end-to-end latency of an active message, in which delivery interrupts the receiving processor, is just over 100 cycles. Since the version of the Alewife software system used in this case study does not offer polling (later versions are expected to), polling was not used. For further details on how message passing is implemented, the reader is referred to the papers by Agarwal et al. [2] and Kranz et al. [16].

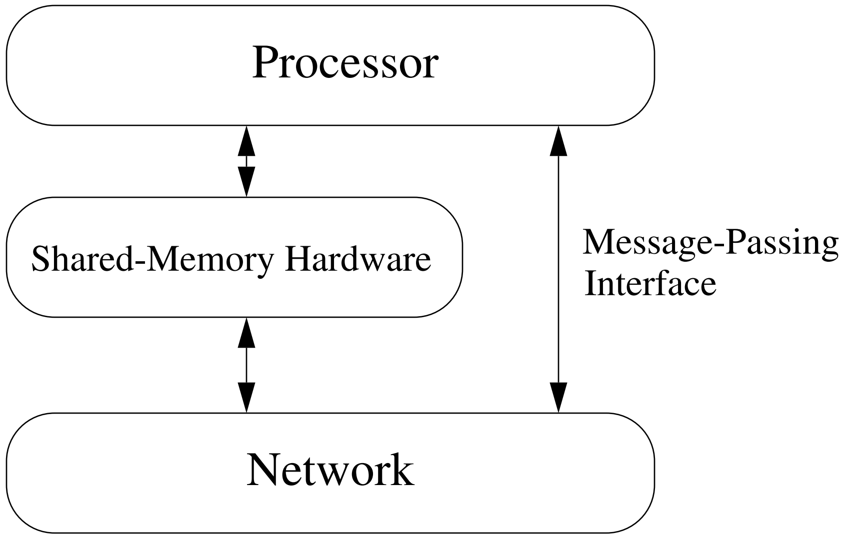

Figure 4.2. Alewife's shared-memory and message-passing interfaces.

4.2 Shared-Memory Implementations

We developed several shared-memory implementations of PBFS, all of which use Mellor-Crummey-Scott queue locks [20]. Our baseline shared-memory implementation (SM) takes the simplest approach by putting the transposition table and the work queues in the shared-memory bank of one processor. To extend a position in the *current* queue, a processor acquires a lock on the *current* queue, extracts a position, releases the lock, and computes the position's extension. Similarly, to add an extension onto the *next* queue, a processor acquires a lock on the *next* queue, adds the extension, then releases the lock. Because the SM implementation centralizes data structures, it performs poorly as the number of processors increases for problem size 6, as shown in Figure 4.3. At 16 and 32 processors, the execution times are about the same.

To attempt to improve the performance of the SM implementation, variations distribute the transposition table across processors' shared-memory banks and privatize the work queues (giving each processor its own set of work queues), distribute load, and prefetch positions. Unless otherwise noted, in all shared-memory implementations, there is one lock per slot of the transposition table, and each processor owns an equal portion of a shared pool of positions. Mutually exclusive access to a particular position is provided by the locks on the transposition table and the work queues. The presence of locks on the

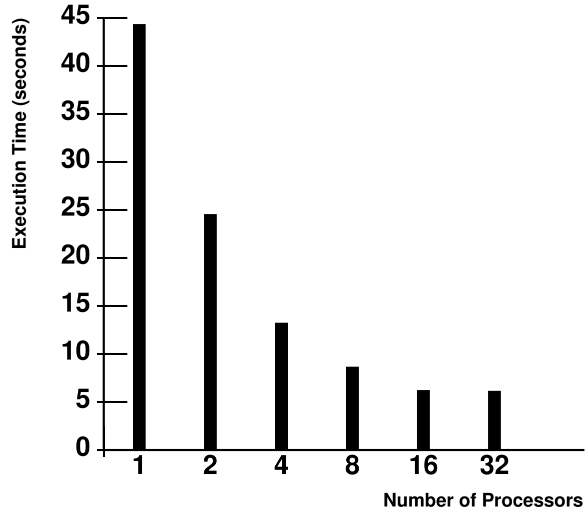

Figure 4.3. Performance of the SM implementation, problem size 6.

four main data structures described in Chapter 3.3 and the placement of these data structures will be described on a case-by-case basis.

4.2.1 Distributed Transposition Table and Privatized Work Queues

The DTABQ ("D" for distributed, "TAB" for transposition table, and "Q" for work queues) implementation attempts to improve the performance of the SM implementation by distributing the transposition table across processors' memory banks and by giving each processor private work queues that are stored in private memory that is local to that processor. Distributing the transposition table means that the slots are interleaved across processors' shared-memory banks, as shown in Figure 4.4. Privatizing the work queues means that each processor has its own *current* queue in private memory and *next* queue in its shared-memory bank. There are no locks on the *current* queues, but the locks on the *next* queues remain. To distribute work, when a processor generates an extension of a position, it first determines if the extension exists in the transposition table. If it does not, the processor computes a destination processor by hashing the bit representation of the extension. Then, by using a shared-memory data transfer, the processor places the extension in the destination processor's *next* queue. Figure 4.5 shows that the DTABQ implementation performs better than the SM implementation for problem size 6 and for all numbers of processors. The largest improvement, 46%, occurs at 32 processors.

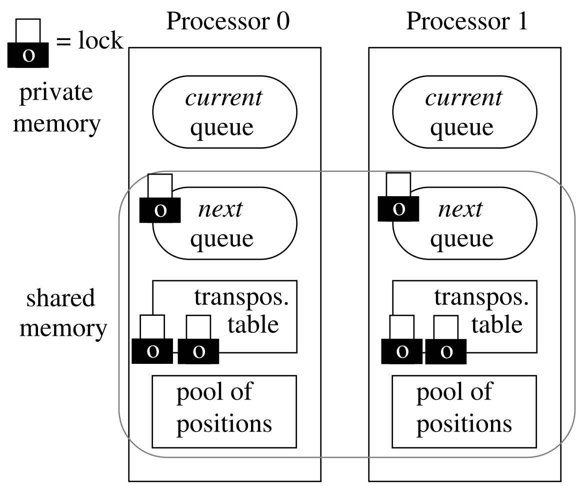

Figure 4.4. Distributed transposition table and privatized work queues in the DTABQ implementation with two processors. Slots of the transposition table are interleaved across processors' shared-memory banks, with one lock per slot. Each processor has its own *current* queue in private memory, and has its *next* queue in shared memory. Each processor has an equal portion of the shared pool of positions.

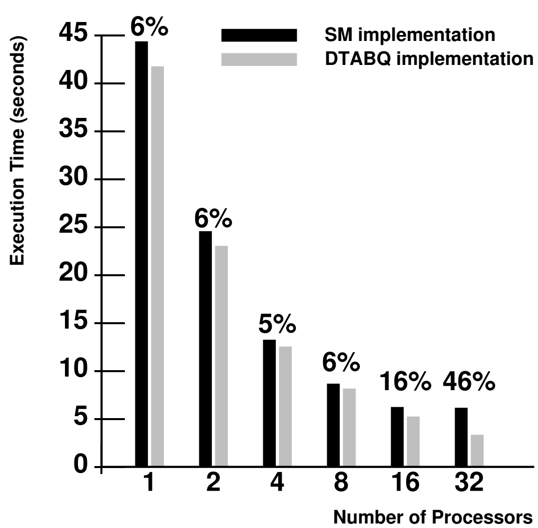

Figure 4.5. Performance of SM and DTABQ implementations, problem size 6. Percentages show how much better the DTABQ implementation is.

4.2.2 Improved DTABQ Implementation

Because the DTABQ implementation shares the transposition table, checking whether an extension resides in the transposition table can involve accesses to remote memory banks. To make accesses to the transposition table local and to reduce the amount of synchronization, the DTABQ implementation can be improved by modifying it in the following manner. Each processor only accesses its own partition of the transposition table. In addition, each processor owns a number of *next* queues equal to the number of processors. Instead of checking whether an extension exists in the transposition table before placing each extension in a *next* queue as in the DTABQ implementation, each processor *unconditionally* places each extension in a *next* queue in a destination processor's set of *next* queues. The particular *next* queue in the set is determined by the number of the processor that generates the extension. Since only one processor writes to any particular *next* queue, no locks are required on these *next* queues. Call this phase in which extensions are generated and placed in processors' sets of *next* queues the *computation phase*. After all extensions have been generated, each processor then determines if extensions in its set of *next* queues are in its partition of the transposition table. If an extension is already in the partition, it is joined with the position in the partition. Call this phase the *checking phase*. Because extensions are unconditionally placed in other processors' sets of *next* queues, accesses to the transposition table during the checking phase are local[2]. In fact, because each processor executes the checking phase on its own set of positions in its *next* queues and on its own partition of the transposition table, no locks are required on the slots of the table. Call this improved implementation the Improved DTABQ implementation.

Figure 4.6 shows that the Improved DTABQ implementation performs better than the DTABQ implementation for eight and more processors, but worse for fewer processors. The reason is that the Improved DTABQ implementation unconditionally places extensions in *next* queues. Thus, it performs more enqueue operations to the *next* queues than the DTABQ implementation does. For problem size 6, the DTABQ implementation performs 113,893 enqueue operations to *next* queues, whereas the Improved DTABQ implementation performs 688,349 such operations. As the number of processors increases, however, the parallelism of these enqueue operations also increases so that with eight and more processors, the Improved DTABQ implementation performs better. In addition,

2. Each slot consists of a counter of positions in the slot and a set of pointers to positions. Thus, accesses to this counter and the *pointers* are local, not accesses to the *bit representations* of positions.

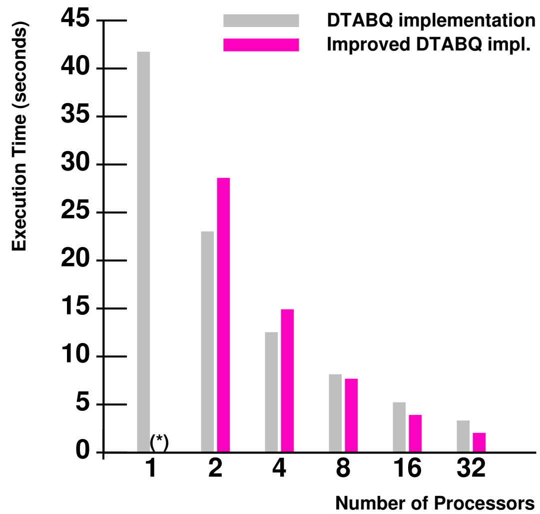

Figure 4.6. Comparison of DTABQ and Improved DTABQ implementations, problem size 6. There is not enough memory in one processor to execute the Improved DTABQ implementation.

Implementation	Number of Processors	Time spent executing LimitLESS software, averaged over all processors (seconds)	Percent of Execution Time
DTABQ	8	0.75	9%
	16	0.70	13%
	32	0.40	12%
Improved DTABQ	8	0	0%
	16	0.0001	0.001%
	32	0.0001	0.005%

Table 4.1. Time spent executing LimitLESS software for DTABQ and Improved DTABQ implementations, problem size 6.

because of remote accesses to the transposition table, the DTABQ implementation spends more time executing LimitLESS software, as shown in Table 4.1.

Number of Processors	Percent of execution time spent idle, averaged over all processors (DTABQ)	Percent of execution time spent idle, averaged over all processors (Improved DTABQ)
2	0.4%	0.7%
4	1.3%	3.1%
8	1.9%	4.2%
16	3.5%	6.1%
32	7.0%	9.7%

Table 4.2. The percentages of total execution time a processor on the average spends idle for the DTABQ and Improved DTABQ implementations. The percentage increases as the number of processors increases because, with a random distribution of the same number of positions among an increasing number of processors, positions are distributed less evenly.

4.2.3 Load Distribution

The DTABQ and Improved DTABQ implementations use a hash function to randomly distribute positions among processors. This random load distribution policy distributes work fairly evenly. As shown in Table 4.2, the percentage of total execution time a processor on the average spends idle is small. Thus, employing more sophisticated scheduling techniques, such as self-scheduling [22], is not warranted.

4.2.4 Prefetching

In our experiments with prefetching, the execution time of the Improved DTABQ implementation can be improved at most 3.8% for problem size 6. We experimented with prefetching data in three places. First, when generating extensions from positions in the *current* queue, processors prefetch one position (its bit representation), its associated solution counter, and a pointer to the next position in the queue. Second, when scanning a slot of the transposition table to check if an extension already exists, processors prefetch one position and its solution counter. Lastly, when dequeuing positions from the *next* queues, processors prefetch one position, its associated solution counter, and a pointer to the next position in the queue. Because of the dynamic, fine-grained nature of the Improved

Number of Processors	Percentage Improvement of Prefetching over No Prefetching
2	1.7%
4	2.2%
8	2.7%
16	3.2%
32	3.8%

Table 4.3. The percentage improvement of prefetching positions from the *next* queues over no prefetching in the Improved DTABQ implementation, problem size 6.

DTABQ implementation, we observed that prefetching positions from the *current* queue and from slots in the transposition table generally performs worse than without prefetching. We believe that positions prefetched from the *current* queue and transposition table are either evicted from direct-mapped caches before they are used or are already present in the cache. However, prefetching positions from the *next* queues does improve performance. We believe the reason is that out of the three places in which prefetching is performed, only when positions are prefetched from the *next* queue do processors access mostly uncached shared data. Recall that another processor most likely placed a given position in the *next* queue and the processor accessing the position has not accessed it before. On the other hand, a position in the *current* queue was accessed by the processor during the checking phase. A position in a slot of the transposition table also can be accessed multiple times during the checking phase, and thus can be cached. Thus, we believe that the conditions just mentioned under which prefetching is ineffective usually do not occur when prefetching positions from the *next* queues.

Table 4.3 shows the percentage improvement of prefetching positions from the *next* queues over no prefetching. As the number of processors increases, the percentage improvement increases because the benefit of prefetching magnifies as parallelism increases. From now on, the Improved DTABQ implementation with prefetching refers to the one that prefetches positions from the *next* queues.

4.3 Message-Passing Implementation

In the message-passing implementation of PBFS, each processor keeps in private memory its own set of the four main data structures. Thus, the transposition table is not shared; each processor accesses its own private partition of the transposition table. Each round in PBFS proceeds as described in Chapter 3.3, except that processors send each extension and the associated solution counter to another processor using one active message with a data payload of 16 bytes. As in the DTABQ implementation, the destination processor is determined by hashing the bit representation of the generated extension. Upon receiving a message, a processor handles the extension in the message in the fashion described in Chapter 3.3. First, if the extension is not in the transposition table, it and its associated solution counter are placed in both the transposition table and the *next* queue. If the extension is already in the transposition table, it is joined with the existing position, whose solution counter is increased by the value of the extension's solution counter. A round ends by passing two barriers. First, a barrier is passed when all extensions have been generated. Second, another barrier is passed when all extensions have been received and processed.

4.4 Comparisons of Shared-Memory and Message-Passing Implementations

Comparing the shared-memory implementations with the message-passing implementation yields the two conclusions of this case study. First, when using distributed shared memory, performing cache coherence actions and decoupling synchronization and data transfer can make a shared-memory implementation less efficient than the message-passing implementation. Second, shared memory offers low-overhead data access and can perform better than message passing for applications that exhibit low contention.

4.4.1 When Message Passing Can Perform Better Than Shared Memory

Recall from Chapter 2 that performing cache coherence actions and decoupling of synchronization and communication can make a shared-memory implementation less efficient than the message-passing one. To illustrate this conclusion, we compare the DTABQ and message-passing implementations. They implement PBFS similarly, and the DTABQ

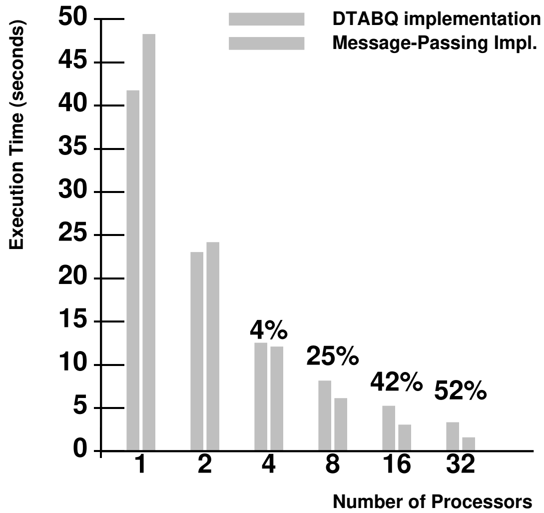

Figure 4.7. Comparison of DTABQ and message-passing implementations, problem size 6. The percentages show how much better the message-passing implementation is than the DTABQ implementation.

implementation is the best shared-memory implementation that uses explicit synchronization. Although the Improved DTABQ implementation does not perform locking operations on any of the four main data structures, has local accesses to the transposition table, and is thus better than the DTABQ implementation for eight and more processors (see Figure 4.6), we defer comparing the Improved DTABQ and message-passing implementations to Chapter 4.4.3.

Figure 4.7 shows that the DTABQ implementation performs worse than the message-passing implementation for problem size 6 and for four and more processors. The message-passing implementation performs up to 52% better than the DTABQ implementation. With one and two processors, the DTABQ implementation performs better. For example, with one processor, the message-passing implementation executes 6.51 seconds longer than the DTABQ implementation. The message-passing implementation for one and two processors performs worse for three reasons. First, the message-passing implementation incurs the end-to-end latency of user-level active messages, whereas the DTABQ implementation does not. Based on the performance statistics given in Chapter 4.1, each active message requires about 100 more cycles than a cached shared-memory access. For one processor, since each position requires one active message, the message-passing implementation spends approximately (100 cycles/position)(688,349 positions)(1 second/ 20,000,000 cycles), or about 3.44 seconds more than the DTABQ implementation in performing data transfer for problem size 6. The other 3.07 seconds of the 6.51 second difference between execution times is spent evaluating the values of data sent per message and copying the data from a message to local variables upon receiving the message. This phenomenon does not significantly affect the difference in execution times as the number of processors increases because the effect of this phenomenon decreases as parallelism of sending messages and shared-memory accesses increases. Second, since there are fewer remote access possibilities for one and two processors than for four or more processors, we observe that the average cache hit ratio in the DTABQ implementation is higher for one and two processors (96.0% and 95.2% for one and two processors, respectively, and 94.0%-94.6% for more processors), thus making the overhead of cache coherence actions lower. Since the difference between cache hit ratios is small, this second reason why the DTABQ performs better than the message-passing implementation does not contribute significantly to the difference in execution times.

4.4.2 When Shared Memory Can Perform Better Than Message Passing

By comparing the DTABQ and message-passing implementations, we saw that message passing can perform better than shared memory. By comparing the SM and message-passing implementations, we will observe that shared memory can perform better than message passing.

Two key insights are the sources of why the SM implementation can perform better than the message-passing implementation. The first key insight is that a given position in problem size 5 has on the average fewer possible moves than a given position in problem size 6 because there are fewer holes and fewer initial pegs in the puzzle board. Table 4.4 illustrates this point by showing the maximum and average number of moves from any given position during each round. Recall that all our implementations except the Improved DTABQ implementation determine whether an extension resides in the transposition table *immediately after* the extension is generated. If an extension is not in the transposition table, it is added to the table and to the *next* queue. Because a given position in problem size 6 has on the average more possible moves than a given position in problem size 5, the rate of adding extensions to the *next* queue will be higher in problem size 6 than in problem size 5 for the SM implementation.

The second key insight is that problem size 6 has a higher average of joined positions per round than problem size 5. In problem size 6, 65.4% of all positions generated per round are joined positions. In contrast, 46.2% of all positions generated per round are joined positions in problem size 5. (Table A.3 and Table A.4 show the average percent-

Round	Problem size 5 maximum number of moves	Problem size 5 average number of moves	Problem size 6 maximum number of moves	Problem size 6 average number of moves
1	3	3.0	4	4.0
2	5	4.3	9	6.3
3	7	4.5	13	7.2
4	7	4.5	15	7.5
5	8	4.6	15	8.0
6	8	4.0	15	8.2
7	6	3.3	16	8.1
8	6	2.6	16	7.9
9	4	1.9	16	7.5
10	3	1.1	14	6.9
11	2	0.5	13	6.2
12	1	0.2	12	5.3
13	0	0	12	4.3
14			11	3.3
15			10	2.4
16			10	1.6
17			4	0.7
18			2	0.4
19			0	0

Table 4.4. Maximum and average number of moves from any given position in a round in problem sizes 5 and 6.

ages per round.) As the transposition table fills up with positions, more positions are joined, and the rate at which positions are added to the *next* queue slows down. Because problem size 6 has a higher percentage of joined positions per round than problem size 5 and because in the SM implementation a processor fetches a new position from the *current* queue if no more extensions can be generated from a given position, the rate of fetching positions from the *current* queue is higher in problem size 6 than in problem size 5.

To illustrate that the rates at which positions are fetched from the *current* queue and added to the *next* queue are greater for problem size 6 than problem size 5, we compare

Number of Processors	SM problem size 6 average request rate for *current* queue divided by problem size 5 request rate	SM problem size 6 average request rate for *next* queue divided by problem size 5 request rate
2	1.92	1.74
4	1.87	1.67
8	2.18	1.99
16	2.55	2.10
32	2.77	2.32

Table 4.5. Ratio of average request rates for work queues for problem size 6 to that for problem size 5 in the SM implementation during the round with the highest number of generated extensions. Since there is no contention when there is only one processor, the ratios are not shown for this machine size.

the request rates of processors to be serviced by the work queues during the round with the highest number of generated extensions (round 8 for problem size 5 and round 11 for problem size 6) in the SM implementation. When a processor is serviced, it performs a series of operations, which we will refer to as one queue operation. A *current* queue operation consists of obtaining a lock, obtaining a position from the *current* queue, and releasing the lock. A *next* queue operation consists of obtaining a lock, adding a position to the *next* queue, and releasing the lock. Table 4.5 shows that average request rate for problem size 6 is 1.67 to 2.77 times the rate for problem size 5, and the ratio of rates generally increases as the number of processors increases. Since a higher request rate implies greater contention, contention for the work queues is definitely higher in problem size 6 than in problem size 5. (Table A.6 and Table A.7 show the actual request rates and the standard deviations of the experiments; ten trials were executed.) We expect higher rates for generally all rounds because both the percentage of joined positions per round and the maximum and average number of possible moves from a given position are higher in problem size 6 than problem size 5.

Lower contention for the work queues in problem size 5 causes less time spent acquiring locks and lower demand on memory access. These factors contribute to why the SM implementation performs better than both the DTABQ and message-passing implementations for problem size 5, as shown in Figure 4.8. Recall that the SM implementation has a

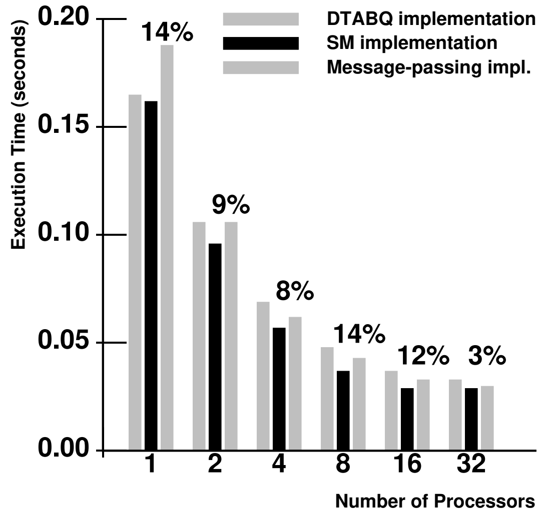

Figure 4.8. Comparison of DTABQ, baseline SM, and message-passing implementations, problem size 5. Percentages show how much better the SM implementation is than the message-passing one.

lock on the *current* queue, whereas the DTABQ implementation does not have locks on the multiple *current* queues. Thus, we should expect the SM implementation to perform worse, but it does not. The reason is that since less time is spent acquiring locks, other overheads are relatively more significant. Recall that the DTABQ implementation hashes the bit representations of extensions in order to distribute work, whereas the SM implementation does not. Because this hashing overhead in the DTABQ implementation is more significant relative to time spent acquiring a lock on the *current* queue in the SM implementation, the DTABQ implementation performs worse. The difference in execution times is not attributed to time spent executing LimitLESS software because the difference between the times the two implementations spend executing LimitLESS software, averaged over all processors, is insignificant relative to execution time.

The SM implementation performs better than the message-passing implementation for all machine sizes for the following reason: data access requests are processed by the underlying hardware *in parallel* with the executing thread on the processor that services the requests, and this process significantly affects execution time when demand on memory access is low. In contrast, in the message-passing implementation, data access requests *interrupt* the executing thread on the servicing processor, thus making processing data access requests and running the executing thread sequential. Thus, we arrive at our second conclusion: shared memory offers low-overhead data access and performs better than message passing for applications that exhibit low contention.

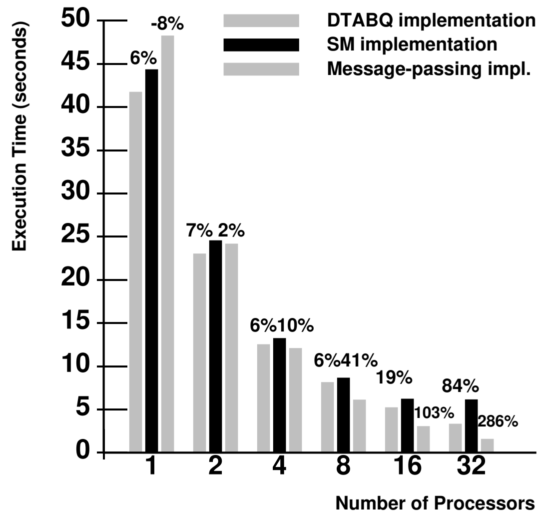

Figure 4.9. Comparison of DTABQ, baseline SM, and message-passing implementations, problem size 6. Percentages show how much worse the SM implementation is than the DTABQ and message-passing implementations.

For problem size 6, this conclusion no longer holds: because contention for the work queues is higher, the demand on memory access is higher. Thus, although shared-memory accesses are being processed in parallel with executing threads, the parallelism is reduced. Figure 4.9 shows that the baseline SM implementation performs worse than the DTABQ and message-passing implementations for problem size 6 for two and more processors. For one processor, the SM implementation performs better than the message-passing implementation because the message-passing implementation incurs the end-to-end latency of user-level active messages and overheads mentioned in the previous section, whereas the SM implementation does not. Again, the overheads in the message-passing implementation do not significantly influence the difference in execution times as the number of processors increases because parallelism of sending messages and shared-memory accesses also increases. For one processor, the performance difference between the SM and message-passing implementations is not as large as that for the DTABQ and message-passing implementations because the SM implementation uses a lock on the *current* queue, whereas the DTABQ implementation does not. The difference in execution times of the SM and DTABQ implementations is not attributed to executing LimitLESS software because for all numbers of processors, both implementations spend about the same time, averaged over all processors, for problem size 6. The SM implementation performs worse than the DTABQ implementation for one processor because more time is spent locking the *current* queue in the SM implementation than hashing positions to distribute work in the DTABQ implementation. Recall that for problem size 5, the SM

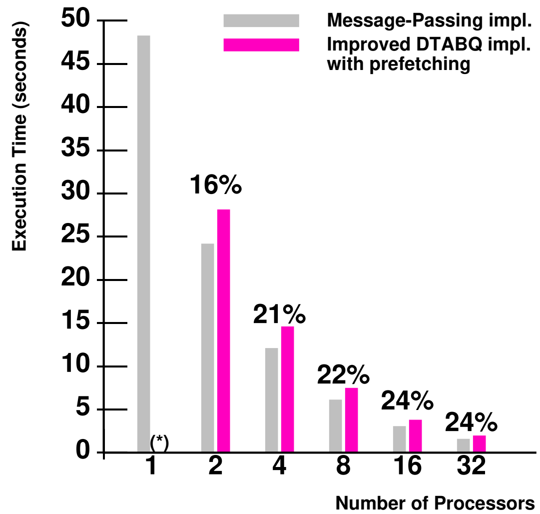

Figure 4.10. Comparison of message-passing and Improved DTABQ implementations, problem size 6. There is not enough memory in one processor (*) to execute the Improved DTABQ implementation. Percentages show how much worse the Improved DTABQ implementation is than the message-passing implementation.

implementation performs better with one processor. The reason is that 34.4% of all explored positions are hashed in the DTABQ implementation for problem size 5, whereas only 16.5% of all explored positions are hashed in the DTABQ implementation for problem size 6. Thus, for one processor, problem size 5 spends more time hashing in the DTABQ implementation than locking in the SM implementation, whereas problem size 6 spends more time locking in the SM implementation than hashing in the DTABQ implementation.

4.4.3 Improved DTABQ and Message-Passing Implementations

When compared with the message-passing implementation, the Improved DTABQ implementation also supports the two conclusions of this thesis. Since it reduces the amount of synchronization by not using any locks on the four main data structures, the Improved DTABQ implementation reduces the negative effect of decoupling synchronization and communication. In addition, since data accesses to a processor's set of *next* queues execute in parallel with the thread that runs on that processor, the Improved DTABQ implementation approaches the performance of the message-passing implementation within 16% in the best case and 24% in the worst case. Figure 4.10 compares the Improved DTABQ and message-passing implementations.

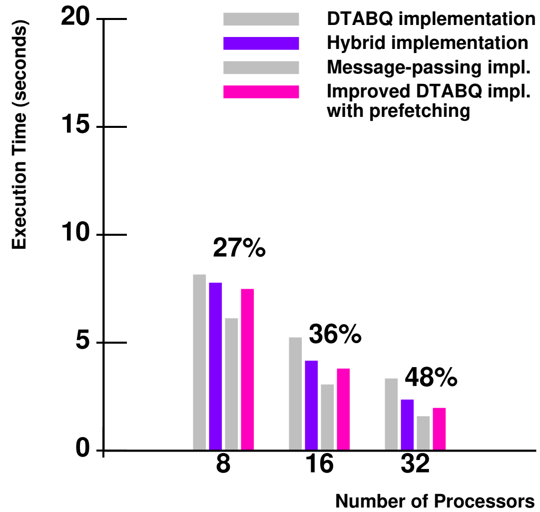

Figure 4.11. Comparison of DTABQ, hybrid, message-passing, and Improved DTABQ implementations, problem size 6. Percentages show how much the hybrid implementation is worse than the message-passing implementation. Because the hybrid implementation requires two pools of positions, it needs at least eight processors to have enough memory to execute.

4.5 Hybrid Implementation

For this application, a hybrid implementation (one that shares the transposition table among processors and uses message passing to distribute extensions) will not perform better than the pure message-passing implementation. Two characteristics of the message-passing implementation support this hypothesis. First, extensions with the same canonical position are sent to the same processor. Second, checking whether these extensions are in this processor's piece of the transposition table always involves accesses to private memory. Because of this locality in the message-passing implementation, a hybrid implementation will perform worse.

We implemented a hybrid implementation that is the same as the DTABQ implementation, with three exceptions. First, extensions are sent as active messages. Second, generating extensions precedes a phase in which it is determined whether extensions reside in the transposition table. Third, the pool of positions is managed differently. Each processor has one pool of positions in its private memory and one pool of positions in its portion of shared memory. Because deadlock may result in the current Alewife implementation when a message handler accesses shared memory, a processor places received extensions in its *next* queue, which refers to positions in the processor's *private* pool of positions.

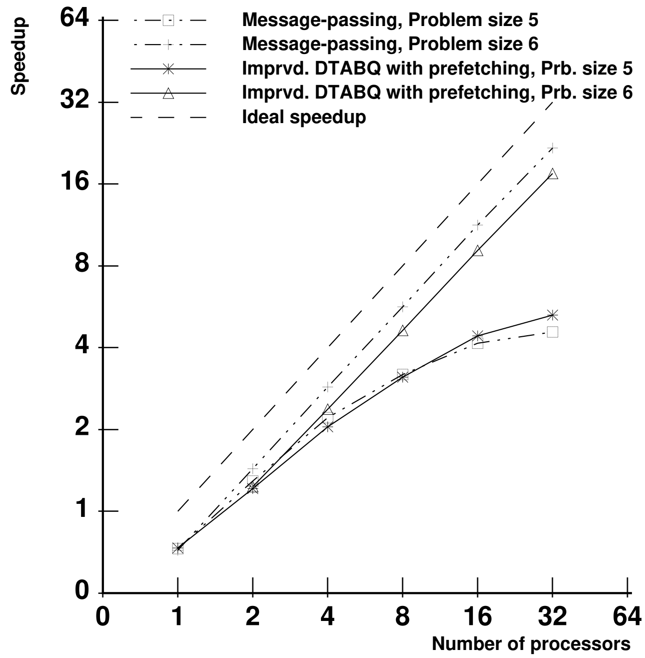

Figure 4.12. Speedups of Improved DTABQ and message-passing implementations over sequential implementation.

Because the transposition table is *shared*, if an extension in the *next* queue does not reside in the table, it is copied to a shared position and placed in the table and *current* queue. Figure 4.11 shows that the hybrid implementation for problem size 6 performs worse than the message-passing implementation, but performs better than the DTABQ implementation. Figure 4.11 also shows that the hybrid implementation performs worse than the Improved DTABQ implementation because the Improved DTABQ implementation has local accesses to the transposition table and has no locks on any of the four main data structures.

4.6 Speedups of the Improved DTABQ and Message-Passing Implementations

Figure 4.12 examines the speedups of the Improved DTABQ and message-passing implementations over a sequential implementation of the breadth-first search algorithm on one node of the Alewife machine. This sequential implementation executes in 0.137 seconds for problem size 5, and executes in 34.694 seconds for problem size 6 (averaged over ten trials). Because of the characteristics of the message-passing implementation mentioned in the previous section and because it does not incur the overhead of performing cache coherence actions, the message-passing implementation generally scales better than the Improved DTABQ implementation. However, for eight and more processors and problem size 5, the Improved DTABQ implementation scales better. This fact supports our second conclusion that with a small problem size and hence lower contention, shared memory performs better. In addition, both implementations scale better for problem size 6 than problem size 5. The reason is that a larger problem size is more suitable for parallel solution because there is dramatically more work to perform (688,349 positions are explored in problem size 6, whereas only 2,463 positions are explored in problem size 5). The average time per processor spent executing LimitLESS software does not affect the scalability of the Improved DTABQ implementation. For both problem sizes and eight and more processors, 2,000 or fewer cycles are spent executing LimitLESS software.

Chapter 5

Related Work

Several papers compare shared-memory and message-passing versions of parallel applications. Chandra et al. [8] compare shared-memory and message-passing implementations of four parallel applications. However, unlike our comparison, which uses an actual machine to make measurements, they simulate a distributed, cache-coherent shared-memory machine and a message-passing machine by using different simulators that run on a CM-5. In addition, their message-passing machine simulator uses Thinking Machines' CMMD message-passing library, which has questionable efficiency. Moreover, their simulators assume constant network latency and do not account for network contention. Martonosi and Gupta [19] compare shared-memory and message-passing implementations of a standard cell router called LocusRoute. However, they use different simulators for their shared-memory and message-passing machines, and run each simulator on a different machine. Kranz et al. [16] evaluate shared-memory and message-passing microbenchmarks on Alewife, and one of their results supports our first main conclusion.

A noteworthy parallel solution by Hatcher and Quinn [15] solves the triangle puzzle on different parallel machines that support either shared memory or message passing in hardware. Their algorithm starts with parallel breadth-first search until each logical processor (a physical processor consists of logical processors) has at least one position to extend, then switches to parallel depth-first search in which each logical processor is responsible for *independently* computing the number of solutions from the positions it owns. Because of the large number of joins encountered when solving the triangle puzzle, implementations that use this approach will duplicate work and thus can perform worse than those that strictly perform parallel breadth-first search and utilize a shared transposition table. Our message-passing implementation in which independent depth-first search begins after each physical processor has at least one position performs worse than the PBFS-only, message-passing implementation on the same machine, a 64-node CM-5. For example, for problem size 5 on 64 processors, performance degrades by a factor of three (setup time included).

The triangle puzzle has been solved using sequential algorithms. Bischoff [4], who won second place in the Internet contest mentioned in Chapter 1, uses algorithmic techniques similar to ours in a sequential algorithm that runs on various machines. On a 486DX2-66, his algorithm that solves problem size 6 runs in 4.13 seconds, whereas our algorithm on the same architecture runs in 7.25 seconds, or 1.8 as slow. The reason Bischoff's sequential algorithm executes faster is that his algorithm stores all reflections of positions in a table, whereas we dynamically compute them; there is a time-space tradeoff. We decided to dynamically compute reflections of positions because physical memory per node on Alewife is low (5MB maximum usable memory) and there is no virtual memory yet.

The triangle puzzle is similar to other tree-search problems, such as the N-queens problem [9, 12, 23, 27, 28], the Hi-Q puzzle [23], and chess [18]. Many of the search techniques, such as exploiting symmetry to reduce the search space and using a transposition table, arise in solving these problems.

Chapter 6

Conclusions

We presented parallel shared-memory and message-passing implementations that solve the triangle puzzle. This thesis is the first controlled case study of message-passing and shared-memory implementations of an application that runs on actual hardware: only the communication interfaces used by the implementations vary; all other system components remain fixed. From this case study, we draw two main conclusions. First, when using distributed shared memory, performing cache coherence actions and decoupling synchronization and data transfer can make a shared-memory implementation less efficient than the message-passing implementation. Second, shared memory offers low-overhead data access and can perform better than message passing for applications that exhibit low contention.

To address the shared memory versus message passing debate, we comment on the performance of our implementations and the ease of programming them. We learned from this case study that under certain circumstances and for this particular application, a shared-memory implementation sometimes performs better than the message-passing one. Under other circumstances, the message-passing implementation performs better. Regarding ease of programming, we found the shared-memory implementations easier to program. One insidious bug (that was fixed) in our message-passing implementation was a race condition that existed when executing code was interrupted by the code in a message handler. Thus, we argue for machines that efficiently support both shared memory and message passing so that all parallel applications can perform well.

Appendix A

Supplemental Tables

Table A.1 and Table A.2 show the execution times for shared-memory and message-passing implementations for problem sizes 5 and 6, respectively. Only execution times that were graphically presented in Chapter 4 are presented. Table A.3, Table A.4, and Table A.5 show the number of positions explored and number of joined positions per round for problem sizes 5, 6, and 7, respectively. Table A.6 and Table A.7 show request rates and standard deviations of work queue operations for problem sizes 5 and 6, respectively; the data from these tables was used to derive the data in Table 4.5.

Implementation	Number of Processors	Execution Time (seconds)	Standard Deviation (seconds)
SM	1	0.162	0.000001
	2	0.096	0.000159
	4	0.057	0.000125
	8	0.037	0.000106
	16	0.029	0.000184
	32	0.029	0.000313
DTABQ	1	0.165	0.000002
	2	0.106	0.000023
	4	0.069	0.000050
	8	0.048	0.000065
	16	0.037	0.000144
	32	0.033	0.000105

Table A.1. Execution times for shared-memory and message-passing implementations, problem size 5.

Implementation	Number of Processors	Execution Time (seconds)	Standard Deviation (seconds)
Improved DTABQ with prefetching	1	0.187	0.000002
	2	0.113	0.000009
	4	0.067	0.000013
	8	0.044	0.000027
	16	0.031	0.000068
	32	0.026	0.000070
Message Passing	1	0.188	0.000001
	2	0.106	0.000014
	4	0.062	0.000038
	8	0.043	0.000075
	16	0.033	0.000159
	32	0.030	0.000061

Table A.1. Execution times for shared-memory and message-passing implementations, problem size 5.

(This portion of Table A.1 is continued from the previous page.)

Implementation	Number of Processors	Execution Time (seconds)	Standard Deviation (seconds)
SM	1	44.370	0.000012
	2	24.577	0.002670
	4	13.260	0.001937
	8	8.679	0.008489
	16	6.245	0.015937
	32	6.165	0.027826
DTABQ	1	41.765	0.000013
	2	23.046	0.003010
	4	12.550	0.001806
	8	8.166	0.004335
	16	5.251	0.003485
	32	3.349	0.009677
Improved DTABQ	1	can't execute	
	2	28.626	0.000394
	4	14.934	0.000358
	8	7.703	0.000265
	16	3.934	0.000135
	32	2.062	0.000090
Improved DTABQ with prefetching	1	can't execute	
	2	28.150	0.000565
	4	14.610	0.000302
	8	7.497	0.000320
	16	3.807	0.000141
	32	1.984	0.000140
Hybrid	1	can't execute	
	2	can't execute	
	4	can't execute	
	8	7.787	0.001534

Table A.2. Execution times for shared-memory and message-passing implementations, problem size 6.

Implementation	Number of Processors	Execution Time (seconds)	Standard Deviation (seconds)
Hybrid	16	4.173	0.000870
	32	2.371	0.001063
Message Passing	1	48.278	0.000013
	2	24.191	0.000853
	4	12.107	0.000599
	8	6.138	0.000478
	16	3.071	0.000244
	32	1.599	0.000230

Table A.2. Execution times for shared-memory and message-passing implementations, problem size 6.

(This portion of Table A.2 is continued from the previous page.)

Round Number	Number of Positions Explored	Number of Joined Positions	Percentage of Joined Positions
1	2	1	50.0%
2	3	0	0%
3	13	2	15.4%
4	49	18	36.7%
5	134	63	47.0%
6	319	187	58.6%
7	524	342	65.3%
8	595	416	69.9%
9	462	323	69.9%
10	265	168	63.4%
11	81	55	67.9%
12	14	8	57.1%
13	1	0	0%
Total	(including initial position) 2463	1614	(average per round) 46.2%

Table A.3. Position statistics per round for problem size 5.

Round Number	Number of Positions Explored	Number of Joined Positions	Percentage of Joined Positions
1	2	1	50.0%
2	4	0	0%
3	25	2	8%
4	166	59	35.5%
5	805	399	50.0%
6	3255	1927	59.2%
7	10842	7250	66.9%
8	29037	21063	72.5%
9	62813	48320	76.9%
10	108651	87704	80.7%
11	144681	121409	83.9%
12	143275	123753	86.4%
13	102976	90417	87.8%
14	53927	47673	88.4%
15	20684	18236	88.2%
16	5893	5132	87.1%
17	1180	1012	85.8%
18	120	92	76.7%
19	12	7	58.3%
Total	(including initial position) 688349	574456	(average per round) 65.4%

Table A.4. Position statistics per round for problem size 6.

Round Number	Number of Positions Explored	Number of Joined Positions	Percentage of Joined Positions
1	2	1	50.0%
2	6	3	50.0%
3	26	3	11.5%
4	234	107	45.7%
5	1362	725	53.2%
6	7320	4571	62.4%
7	32694	22460	68.7%
8	124336	92116	74.1%
9	395100	308524	78.1%
10	1057990	861074	81.4%
11	2372103	1994427	84.1%
12	4435444	3826796	86.3%
13	6881092	6058173	88.0%
14	8829772	7897726	89.4%
15	9336725	8452864	90.5%
16	8102692	7401676	91.3%
17	5755731	5289176	91.9%
18	3356026	3094497	92.2%
19	1602445	1479590	92.3%
20	618748	570793	92.2%
21	191239	175627	91.8%
22	47219	43046	91.2%
23	8620	7762	90.0%
24	1061	908	85.6%
25	144	120	83.3%
26	0	0	0%
Total	(including initial position) 53158132	47582765	(average per round) 73.7%

Table A.5. Position statistics per round for problem size 7.

Number of Processors	Request rate for *current* queue (operations/ cycle)	Standard deviation for request rate for *current* queue (operations/ cycle)	Request rate for *next* queue (operations/ cycle)	Standard deviation for request rate for *next* queue (operations/ cycle)
2	5066.86	48.86	5040.81	139.40
4	5707.26	146.01	5725.38	518.98
8	6684.33	358.28	6621.06	1119.82
16	9702.18	916.56	9382.08	2145.33
32	18249.81	3294.98	16756.77	5046.32

Table A.6. Request rates and standard deviations for work queue operations, problem size 5.

Number of Processors	Request rate for *current* queue (operations/ cycle)	Standard deviation for request rate for *current* queue (operations/ cycle)	Request rate for *next* queue (operations/ cycle)	Standard deviation for request rate for *next* queue (operations/ cycle)
2	9724.42	161.89	8753.54	166.60
4	10647.27	163.51	9583.73	180.65
8	14597.15	1535.84	13157.04	1522.94
16	24781.83	16010.10	19662.13	4050.93
32	50469.07	38950.69	38912.12	3363.33

Table A.7. Request rates and standard deviations for work queue operations, problem size 6. The standard deviations for 16 and 32 processors for the request rate for *current* queue operations are high because the processor whose memory bank contains the work queues has a request rate that is much higher than all other processors. The reason is that all accesses to the work queues for this processor are local. For all other processors, the request rates are equal to about the average shown.

Bibliography

[1] D. Applegate, G. Jacobson, and D. Sleator. "Computer Analysis of Sprouts." Carnegie Mellon University, School of Computer Science, TR 91-144, May 1991.

[2] 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." Submitted for publication, December 1994.

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors." *IEEE Micro*, June 1993, pp. 48-61.

[4] M. Bischoff. "Approaches for Solving the Tri-Puzzle." November 1993. Available via anonymous ftp from `lucy.ifi.unibas.ch` as `tri-puzzle/michael/doku.ps`.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer." In *Proceedings of the Twenty-first Annual International Symposium on Computer Architecture*, Chicago, Illinois, April 1994. pp. 142-153.

[6] J. Carter, J. Bennett, W. Zwaenepoel. "Implementation and Performance of Munin." In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, California, October 1991. pp. 152-164.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991, pp. 224-234.

[8] S. Chandra, J. Larus, and 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.

[9] P.-C. Chen. *Heuristic Sampling on Backtrack Trees*. Ph.D. thesis. Stanford University. May 1989. (Also available as Stanford Technical Report CS 89-1258.)

[10] L. Crowl, M. Crovella, T. LeBlanc, M. Scott. "Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search." University of Rochester, Dept. of Computer Science, TR 451, April 1993.

[11] W. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, 1987.

[12] R. Finkel and U. Manber. "DIB--A Distributed Implementation of Backtracking." *ACM Transactions on Programming Languages and Systems*, April 1987, Vol. 9, No. 2. pp. 235-256.

[13] J. Gittinger; T. Chikayama and K. Kumon. Proofs that there are no solutions for the triangle puzzle for size 3*N*+1. Available via anonymous ftp from `lucy.ifi.unibas.ch` in directories `tri-puzzle/gitting` and `tri-puzzle/jm`. November 1993.

[14] S. Gutzwiller and G. Haechler. "Contest: How to Win a Swiss Toblerone Chocolate!" August 1993, Usenet newsgroup `comp.parallel`.

[15] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press: 1991.

[16] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. "Integrating Message-Passing and Shared-Memory: Early Experience." In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993. pp. 54-63.

[17] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. "The Stanford FLASH Multiprocessor." In *Proceedings of the Twenty-first International Symposium on Computer Architecture*, Chicago, Illinois, April 1994. pp. 302-313.

[18] B. Kuszmaul. *Synchronized MIMD Computing*. Ph.D. thesis. MIT Laboratory for Computer Science. May 1994.

[19] 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.

[20] J.M. Mellor-Crummey and M.L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." *ACM Transactions on Computer Systems*, February 1991, Vol. 9, No. 1. pp. 21-65.

[21] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[22] C. Polychronopoulos and D. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers." *IEEE Transactions on Computers*, December 1987. pp. 1425-1439.

[23] E. Reingold, J. Nievergelt, N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, 1977.

[24] C. Seitz. "Concurrent VLSI Architectures." *IEEE Transactions on Computers*, December 1984. pp. 1247-1265.

[25] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary.* Nov. 1993.

[26] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. "Active Messages: A Mechanism for Integrated Communication and Computation." In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. pp. 256-266.

[27] C.K. Yuen and M.D. Feng. "Breadth-First Search in the Eight Queens Problem." *ACM SIGPLAN Notices*, Vol. 29, No. 9, Sep. 1994. pp. 51-55.

[28] Y. Zhang. *Parallel Algorithms for Combinatorial Search Problems*. Ph.D. thesis. UC Berkeley. November 1989. (Also available as UC Berkeley Computer Science Technical Report 89/543.)