Synchronized MIMD Computing

by

Bradley C. Kuszmaul

S.B. (mathematics), Massachusetts Institute of Technology (1984)
S.B. (computer science and engineering), Massachusetts Institute of Technology (1984)
S.M. (electrical engineering and computer science), Massachusetts Institute of Technology (1986)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 1994

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
F. R. Morgenthaler
Chair, Department Committee on Graduate Students

Synchronized MIMD Computing
by
Bradley C. Kuszmaul

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1994, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Fast global synchronization provides simple, efficient solutions to many of the system problems of parallel computing. It achieves this by providing composition of both performance and correctness. If you understand the performance and meaning of parallel computations A and B, then you understand the performance and meaning of "A; barrier; B".

To demonstrate this thesis, this dissertation first describes the architecture of the Connection Machine CM-5 supercomputer, a synchronized MIMD (multiple instruction stream, multiple data stream) computer for which I was a principal architect. The CM-5 was designed to run programs written in the data-parallel style by providing fast global synchronization. Fast global synchronization also helps solve many of the system problems for the CM-5, including clock distribution, diagnostics, and timesharing.

Global barrier synchronization is used frequently in data-parallel programs to guarantee correctness, but the barriers are often viewed as a performance overhead that should be removed if possible. By studying specific mechanisms for using the CM-5 data network efficiently, the second part of this dissertation shows that this view is incorrect. Interspersing barriers during message sending can dramatically improve performance of many important message patterns. Barriers are compared to two other mechanisms, bandwidth matching and managing injection order, for improving the performance of sending messages.

The last part of this dissertation explores the benefits of global synchronization for MIMD-style programs, which are less well understood than data-parallel programs. To understand the programming issues, I engineered the StarTech parallel chess program. Computer chess is a resource-intensive irregular MIMD-style computation, providing a challenging scheduling problem. Global synchronization allows us to write a scheduler which runs unstructured computations efficiently and predictably. Given such a scheduler, the run time of a dynamic MIMD-style program on a particular machine becomes simply a function of the critical path length C and the total work W. I empirically found that the StarTech program executes in time $T \approx 1.02W/P + 1.5C + 4.3$ seconds, which, except for the constant-term of 4.3 seconds, is within a factor of 2.52 of optimal.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Contents

List of Figures

Chapter 1

Introduction

The Synchronized MIMD Thesis

Fast global synchronization can solve many of the system problems of parallel computing.

To demonstrate that fast global synchronization solves many of the system problems of parallel computing, this dissertation first describes hardware and then software, both of which are organized around fast global synchronization. Fast global synchronization is not the only way to solve the system problems of parallel computing, but it provides simple solutions to them. This thesis describes in detail the Connection Machine CM-5 supercomputer and the StarTech massively parallel chess program, treating these systems as example points in the design space of parallel systems. I explain the systems, and rather than systematically comparing various ways to solve the problems, I show simple solutions to problems that have been difficult to solve previously.

The Connection Machine CM-5: Architectural Support for Data-Parallel Programming.

The idea of using fast global synchronization to solve system problems grew out of the CM-5 project which started in 1987 at Thinking Machines Corporation. The primary design goal of the CM-5 was to support the *data-parallel* programming model [HS86, Ble90]. Data-parallel programs run efficiently on the Connection Machine CM-2 computers, which is a *SIMD* (single-instruction stream, multiple-data stream) machine. It was of prime importance for any new machine to continue to run such programs well. As one of the principal architects of the CM-5,[1] I helped design a *MIMD* (multiple-instruction stream, multiple-data stream) machine to execute data-parallel programs.[2]

The data-parallel style of programming is successful because it is simple and programmers can

[1]I was the first member of the CM-5 design team, which eventually grew to include Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. All told, about 500 people participated in the implementation of the CM-5.

[2]The SIMD/MIMD terminology was developed by Flynn [Fly66].

reason about it. A data-parallel program is an ordinary serial program with vector primitives.[3] Vector primitives include

- elementwise operations,
- bulk data communications operations, and
- global reductions and scans.

In the following, we denote vectors by upper-case letters and scalars by lower case letters. Elementwise operations are exemplified by vector addition, which can be expressed as $A \leftarrow B + C$ (which means that for each j, $B[j] + C[j]$ is stored into $A[j]$.) Bulk data communications can be expressed, for example, as $A[I] \leftarrow B$ (which means that each j, $B[j]$ is stored into $A[I[j]]$.) Global reductions and scans can be expressed, for example as `GLOBAL_OR(`A`)` (which returns the logical 'or' of all the elements of A.) In addition, data-parallel programming languages typically provide a conditional execution construct, the `WHERE` statement, that looks like this:

`WHERE` (*expression*)
    body.

In the body of the `WHERE`, all vector operations are conditionally executed depending on the value of the expression. For example

`WHERE` $(B < 0)$
    $B \leftarrow B + C$;

has the effect of incrementing $B[j]$ by $C[j]$ for each j such that $B[j] < 0$.

Data parallel programs have traditionally been run on SIMD machines. In fact, the SIMD machines engendered the data-parallel style of programming [Chr83, Las85, HS86, Ble90]. Examples of SIMD machines include the Illiac-IV [BBK*68], the Goodyear MPP [Bat80], and the Connection Machine CM-1 and CM-2 [Hil85]. Such machines provide a collection of processors and their memories that are controlled by a front-end computer (see Figure 1-1.) The front-end broadcasts each instruction to all of the processors, which execute the instruction synchronously. The broadcast network is embellished with an 'or' network that can take a bit from every processor, combine them using logical or, and deliver the bit to the front end (and optionally to all the processors.) The data network allows data to be moved, in parallel, between pairs of processors by sending messages from one processor to another. The data network is also synchronously controlled by the front-end, which can determine when all messages in the data network have been delivered. The SIMD architecture is synchronous down to the clock cycle.

To execute a data parallel program on a SIMD machine is straightforward. The SIMD machine has a single instruction counter, which matches the program's single thread of control. The vectors are distributed across the machine. The program is executed on the front-end computer with each standard serial statement executed as for a serial program. To execute a vector elementwise operation, the instructions encoding the operation are broadcast, and each processor manipulates its part of the vectors. To execute bulk communications operations, instructions are broadcast to move the data through the data network of the machine. To execute a global scan, instructions are

[3] Examples of of SIMD vector primitives include PARIS, the Parallel Instruction Set developed for the CM-2 [Thi86b], and CVL, a C vector library intended to be portable across a wide variety of parallel machines [BCH*93].

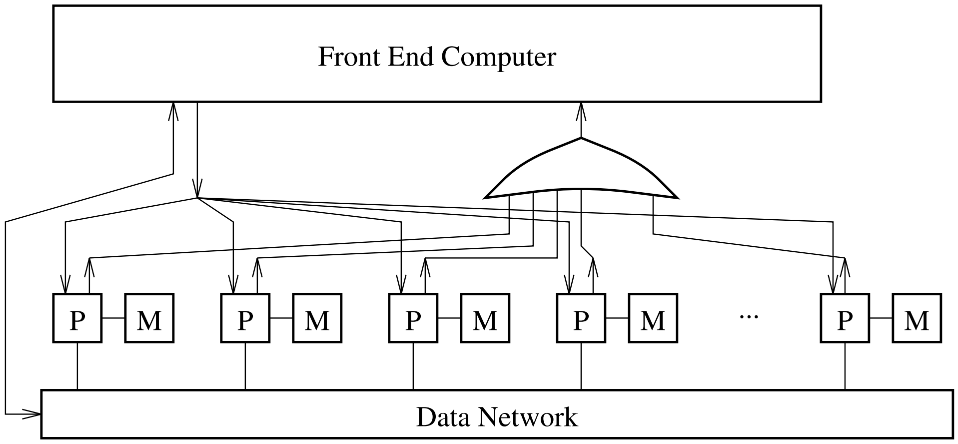


Figure 1-1: The organization of a SIMD computer. The front-end computer controls the processor-memory pairs, through a broadcast network, on a clock-cycle by clock-cycle basis. The 'or' network accepts a bit from each processor and delivers the logical 'or' of all the bits to the front-end. The front-end also controls the data network.

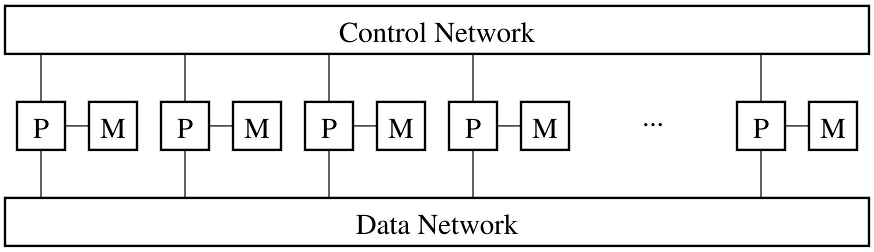


Figure 1-2: The organization of a synchronized MIMD computer. The processors are interconnected by two networks: a data network and a control network.

broadcast to implement a parallel-prefix reduction. The execution of a reduction is similar to a scan, except that the reduced value is communicated to the host through the global 'OR' network. To execute conditional operations, a context mask is maintained in each processor. Each processor conditionally executes the broadcast instructions based on this mask. The `WHERE` statement simply manipulates the context mask.

In the CM-5, we departed from the SIMD approach and built a *synchronized MIMD* machine. A synchronized MIMD machine consists of a collection of processors, a data network, and a control network (see Figure 1-2.) The processors' job is to perform traditional operations on local data, e.g., floating point operations. The data network's job is to move data from one processor to another via message passing. The control network's job is to synchronize an entire set of processors quickly, and to implement certain multiparty communication primitives such as broadcast, reduction, and parallel prefix.

To execute a data parallel program on a synchronized MIMD machine, we simulate the SIMD machine. If a synchronized MIMD machine can simulate a SIMD machine efficiently enough, then we can use the synchronized MIMD machine to execute both MIMD-style and data-parallel computations, instead of using different machines for different styles of computation. A SIMD computation can be simulated on a synchronized MIMD machine by transforming the serial SIMD program that runs on the front-end of the SIMD machine into a program that runs in parallel on every processor of the synchronized MIMD machine. The vectors are laid out across the machine as for a SIMD machine. Each vector primitive is implemented as a subroutine that performs the 'local' part of the primitive for a processor. The 'serial' part of the data-parallel code executes redundantly on every processor, calling the subroutines to perform the local part of each vector primitive.[4] Bulk data transfers are accomplished using the data network. Global scans and reductions use the control network. For conditional operations a context mask is maintained, just as for SIMD machines. To keep the processors in step, the machine is globally synchronized between nearly every vector primitive, which justifies the hardware for a control network.[5]

Without frequent global synchronization the program would execute incorrectly. Consider the following code fragment:

(R1)	$A[I] \leftarrow B;$
(R2)	$A \leftarrow A + C;$

Line R1 calls for interprocessor communication, and then Line R2 uses the result of the communication, A, as the operand to a vector multiply, and then stores the result back into A. Consider what happens if we do not insert a synchronization between Lines R1 and R2. Processor 0 might finish sending its local parts of B to the appropriate places in A, and then Processor 0 could race ahead to start executing Line R2. There would be no guarantee that the local copy of A had been completely updated, however, since some other processor might still be sending data to Processor 0, so the values provided to the vector addition could be wrong. To add insult to injury, after doing the vector multiply, the data being sent to Processor 0 data could then arrive and modify the local part

[4]The idea of distributing a single program to multiple processors has been dubbed "SPMD," for *single-program, multiple data* [DGN*86]. H. Jordan's language, The Force, was an early SPMD programming language dating from about 1981 [JS94] and appearing a few years later in the literature [Jor85, Jor87, AJ94]. S. Lundstrom and G. Barnes describe the idea of copying a program and executing it on every processor of the Burroughs Flow Model Processor (FMP) [LB80]. Here, however, we focus on how to execute a data-parallel program rather than how to generally program in a SPMD style. In Chapter 4 we will consider the problem of running more general MIMD programs.

[5]The idea of building a MIMD machine with a synchronization network is not original with the CM-5. The Burroughs Flow Model Processor (FMP), proposed in 1979, included a control network and a data network that connected processors to memories [LB80]. The barrier network of the FMP was a binary tree that could synchronize subtrees using split-phase barriers. The proposed method of programming the FMP was to broadcast a single program to all the processors, which would then execute it, using global shared memory and barrier synchronization to communicate. The FMP was not built, however. The DADO machine [SS82] of S. Stolfo and D. Shaw provides a control network for a SIMD/MIMD machine. In SIMD mode, the control network broadcasts instructions, while in MIMD mode a processor is disconnected from 'above' in the control network so that it can act as the front-end computer for a smaller SIMD machine. The DADO machine performs all communication in its binary-tree control network. The DATIS-P machine [PS91] of W. Paul and D. Scheerer provides a permutation network and a synchronization network. The DATIS-P permutation network provides no way of recovering from collisions of messages. Routing patterns must be precompiled, and synchronization between patterns is required to ensure the correct operation of the permutation network. The CM-5 design team developed the idea of using split-phase global synchronization hardware [TMC88], and carried it to a working implementation. In independent work, C. Polychronopoulos proposed hardware that would support a small constant number of split-phase barriers that each processor could enter in any order it chose [Pol88]. R. Gupta independently described global split-phase barriers in which arbitrary subsets of processors could synchronize, using the term *fuzzy barrier* [Gup89]. Gupta's proposed implementation is much more expensive than a binary tree. The term *split-phase* describes the situation more accurately than does the term *fuzzy*. M. O'Keefe and H. Dietz [OD90] discuss using hardware barriers that have the additional property that all processors exit the barrier simultaneously. Such barriers allow the next several instructions on each processor to be globally scheduled using, for example, VLIW techniques [Ell85].

of A, overwriting the result of the vector addition. The easiest way to solve this problem is to place a barrier synchronization between Lines R1 and R2.

In *barrier synchronization,* a point in the code is designated as a barrier. No processor is allowed to cross the barrier until all processors have reached the barrier.[6] One extension of this idea is the *split-phase barrier,* in which the barrier is separated into two parts. A point in the code is designated as the *entry-point* to the barrier, and another point is designated as the *completion-point* of the barrier. No processor is allowed to cross the completion point of the barrier until all processors have reached the entry-point. Split-phase barriers allow processors to perform useful work while waiting for the barrier to complete.[7]

Barriers that synchronize only the processors are inadequate for some kinds of bulk communication. During the execution of Line R1 above, each processor may receive zero, one, or more messages. Every processor may have sent all its messages, but no processor can proceed to execute Line R2 until all its incoming messages, some of which may still be in the network, have arrived. To address this problem, the CM-5 provides a *router-done* barrier synchronization that informs all processors of the termination of message routing in the data network.

The router-done barrier is implemented using *Kirchhoff counting* at the boundary of the data network. Each network interface keeps track of the difference between the number of messages that have arrived and the number that have departed. Each processor notifies the network interface when it has finished sending messages, and then the control network continuously sums up the differences from each network interface. When the sum reaches zero, the "router-done" barrier completes.[8]

Kirchhoff counting has the several advantages over the other approaches to computing router-done.[9] Irrelevant messages (such as operating-system messages) can be ignored, allowing router-done to be computed on user messages only. Kirchhoff counting can detect lost or created messages, because the sum never converges to zero. It is independent of the topology of the data network. Finally, Kirchhoff counting is fast, completing in the same time as it takes for a barrier, independently of the congestion in the data network.

Even with hardware support to execute global synchronization quickly, we would like to avoid doing more synchronization than we need. Each synchronization costs processor cycles to manipulate the control network, and also costs the time it takes for all the processors to synchronize. If one processor has more work to do than the others, that processor makes the others wait. This inefficiency is related to the inefficiency of executing conditional instructions on a SIMD machine, since in both cases processors sit idle so that other processors can get work done.

There are several ways to further reduce the cost of synchronization, including removing synchronization and weakening the synchronization. To remove synchronization we can observe that not every vector primitive requires a synchronization at the end of the operation. Synchronization is only required when processors communicate. To weaken synchronization, we can observe that processors could do something useful while waiting for synchronization to complete.

---

[6] B. Smith credits H. Jordan with the invention of the term "barrier synchronization" [Smi94]. According to Smith, Jordan says that the name comes from the barrier used to start horse races. Jordan used barriers to synchronize programs for the Finite Element Machine described in [Jor78]. Smith states that Jordan later used the idea in The Force, an early SPMD programming language. Other descriptions of barriers can be found in [TY86, DGN*86].

[7] Hardware for split-phase barriers was designed for the proposed FMP [LB80]. R. Gupta proposed split-phase barriers in which arbitrary subsets of processors could synchronize, using the term *fuzzy barrier* [Gup89].

[8] The name "Kirchhoff counting" is related to Kirchhoff's current law (see, for example, [SW75]).

[9] Other approaches to computing router-done include software and hardware. In software one can acknowledge all messages, and then use a normal processor-only barrier. Such an approach can double the message traffic, or requires modifications to the data network. [PC90] The Connection Machine CM-1 uses a hardware global-or network to determine when the router is empty [Hil85].

(D1) $A \leftarrow B \cdot C;$
(D2) $D \leftarrow B + E;$

Figure 1-3: A fragment of data-parallel code.

(N1) for $i \in$ `localindices()`
(N2) $\quad A_l[i] \leftarrow B_l[i] \cdot C_l[i];$
(N3) `barrier();`
(N4) for $i \in$ `localindices()`
(N5) $\quad D_l[i] \leftarrow B_l[i] + E_l[i];$
(N6) `barrier();`

Figure 1-4: The naive translation, of the data-parallel code fragment, into the local code to run on a synchronized MIMD processor. We use the notation A_l to denote the part of array A that is kept locally on the processor. The value of `localindices()` is the set of indices of the arrays that are kept on a single processor (we assume here that the arrays are all aligned.) The code updates the local part of A, performs a barrier, updates the local part of D, and performs a barrier.

(O1) for $i \in$ `localindices()`
(O2) $\quad t \leftarrow B_l[i];$
(O3) $\quad A_l[i] \leftarrow t \cdot C_l[i];$
(O4) $\quad D_l[i] \leftarrow t + E_l[i];$
(O5) `barrier();`

Figure 1-5: The optimized synchronized MIMD code. We removed the barrier on Line (N3), collapsed the loops, and performed common subexpression analysis to avoid loading $B_l[i]$ twice from memory.

One can remove barriers between statements that have only local effects. For example, if we have the data parallel code shown in Figure 1-3, and if we assume that the arrays are all the same size and are aligned so that $A[i]$, $B[i]$, $C[i]$, $D[i]$, and $E[i]$ are all on the same processor, then the naive per-processor code would look like Figure 1-4. We observe that the barrier on Line (N3) is not needed because there are no dependencies between Lines (D1) and (D2) in the original code. We can also collapse the loops so that only one pass is needed, and we can avoid loading the value of $B[i]$ twice, resulting in the code of Figure 1-5. By transforming the code containing synchronizations, we have not only reduced the amount of synchronization, but we have exposed additional opportunities for code optimization.

Removing barriers from code that has only local effects is straightforward, but the situation is more complex when there is interprocessor communication. For example, when executing a *send*, expressed as $A[I] \leftarrow B$, a synchronization is required after the operation to make sure all the updates have taken place. When performing a *get*, expressed as $B \leftarrow A[I]$, a synchronization is required before the operation to make sure that the value of A is globally up-to-date before fetching the data. One simple rule is to include a barrier before and after any code that communicates.

Often, simply expressing a barrier as a split-phase barrier makes an optimization obvious. Figure 1-6 shows how a while loop controlled by a global ‘sum’ operation might be translated into

(M1) while (`global_sum`(v)) (M2) ...	$\Rightarrow$	(S1) `start_global_sum`(v); (S2) while (`complete_global_sum`()) (S3) ... (S4) `start_global_sum`(v);

Figure 1-6: By transforming the monolithic global 'sum' operation on the left into the split-phase global 'sum' operations on the right, we can expose opportunities to hide the latency of the global 'sum' operation.

split-phase operations.[10] First, the global sum operation is initialized, providing the value of `v` to the control network. Then, the global sum is completed and the result is used to control the while loop. Before the next iteration of the while loop, the next global sum is started. Now, we have the opportunity to start the global 'sum' on Line (S4) sooner. We could move Line (S4) to just after the last modification of v in the elided code of Line (S3). By transforming the code, we can hide the latency of the global synchronization.

Sometimes barriers should not be removed or weakened. A split-phase barrier does not provide the performance composition property provided by a monolithic barrier. To understand the performance of

$$A\text{; enter-barrier; } B\text{; complete-barrier; } C\text{;}$$

requires that you understand how A and B can interact and how B and C can interact.[11] This is more difficult to understand than the case

$$A\text{; barrier; } B\text{;}$$

where the the two parts of the program are completely separated by a monolithic barrier. We will see several examples in this dissertation where barriers are not needed for the correct execution of a program, but are needed to achieve high performance. In Chapter 3 we show that barriers can improve the performance of interprocessor communication, and in Chapter 5 we show that barriers are useful in a dynamic-MIMD program to keep idle processors from swamping busy processors.

Thus, fast global synchronization hardware allows straightforward and efficient execution of data parallel programs on MIMD machines. It also turns out that by using global synchronization as the underlying organizing strategy of the CM-5, we were also able to solve many other system problems of parallel computing such as time-sharing, diagnostics, and clock distribution.

We also used the global synchronization mechanism to support the operating system in the CM-5. The timesharing system globally schedules the processors so that they are all running the same user program at the same time.[12] We treat the messages in the network as part of the process state, and at the time of a process-switch, the entire network state is saved in the process descriptor. Because the network state is swapped out, the performance of each process is independent of the

[10]One can perform a split-phase scan or reduction using the CM-5 control network. Such an operation implicitly includes a split-phase barrier, since the reduction does not complete until all processors have provided a value.

[11]Before the split phase barrier completes, some processors may be executing A, and some may be executing B. After every processor has entered the barrier and the barrier has completed, some of the processors may still be executing B while others have started to execute C.

[12]The CM-5 is also divided into partitions with each partition operating independently. Here we focus on the behavior of a single partition.

behavior of other processes. In order to be able to quickly empty the data network and to evenly distribute the network state among the processors, we developed the *all-fall-down* mechanism, in which all the messages in the network are immediately delivered to a nearby processor.

This context-switching decision also solves a problem with user-level messages. To provide high-performance protected network access to the user, the processor-network interface includes address-translation hardware, which allows user-mode code to be given direct access to the interface. Thus messages can be directly sent from user-code in one processor to user-code in another, without giving the user access to any other process. No operating-system call is required. Since the user has direct access to the network, the user also must take care not to deadlock the network. It does not hurt the operating system if the user deadlocks the network, since the operating system empties the network on every context-switch. Deadlocking the network has the same effect on the user as if the user had written an infinite loop.[13]

How to Get Good Performance from the CM-5 Data Network

Compiling data-parallel programs for the CM-5 introduces many opportunities for optimizing the performance of the code. Optimizing the code sometimes turns out to be tricky, however. Steve Heller of Thinking Machines Corporation had noticed that a sequence of cyclic shifts separated by barriers actually runs faster than if the barriers are removed [Hel92]. I had expected such a communications operation to run faster without the barriers, since up to that point I generally thought of barriers as overhead that should be removed if possible. Thus, Heller's observation was surprising. A study was embarked on, at MIT, to to understand how best to program the data network of a machine such as the CM-5.[14]

We concluded that programmers of the Connection Machine CM-5 data network can improve the performance of their data movement code more than a factor of three by selectively using global barriers, by limiting the rate at which messages are injected into the network, and by managing the order in which they are injected. Barriers eliminate target-processor congestion, and provide a kind of bulk end-to-end flow control that allows the programmer to schedule communications globally. Injection-reordering improves the statistical independence of the various packets in the network at any given time. Barriers and tuned injection rates provide forms of flow control. Although we only experimented with the CM-5, we expect these techniques to apply to other parallel machines.

Why do barriers speed up a sequence of cyclic shifts? We demonstrated that, without any barriers, some processors fall behind the others. The work from later phases of the computation systematically interferes with uncompleted work from earlier phases. We are able to see what was going on because the CM-5 processors have access to a clock that is globally consistent to within a single 33 megahertz clock cycle. We recorded, for each message, the time it is injected into the network and the time it is delivered. For cyclic shift, the receivers are the bandwidth bottleneck, so it is important to keep all the receivers busy. We saw with our timestamping experiment, however, for hundreds of thousands of cycles at a stretch, that some receivers had no messages waiting to be delivered, and some receivers had over ten messages waiting to be delivered. The imbalance among receivers systematically gets worse over time. By periodically synchronizing all the processors,

[13]The CM-5 makes it easy, however, to implement protocols such as remote-fetch without deadlocking and without incurring a large bookkeeping overhead. The CM-5 accomplishes this by providing two independent networks that comprise the data network.

[14]Eric A. Brewer and I performed the performance study. Brewer, with R. Blumofe, was developing a CM-5 communications library called Strata [BB94b]. Strata's goal is to improve the programming interface and performance of message passing as compared to Thinking Machines' CMMD [Thi93] and Berkeley's CMAM [vCG*92].

the imbalance among receivers is removed, allowing the machine to operate at peak performance. Barriers improve the performance of a sequence of cyclic shifts by a factor of 2 to 3. Thus, not only is global synchronization useful for writing *correct* programs, but it can also help us to improve and to understand the performance of a program.

We knew that the order in which messages are injected could affect the performance of the data network. For example, a bad way to implement all-to-all communication is for every processor to send a block of messages to Processor 0, then to have every processor send a block to Processor 1, and then to Processor 2, and so on. That order dramatically reduces the performance of the program, because there are always only a few processors receiving the data that all the processors are trying to send. We had hypothesized that if on step i, processor j sent a block of messages to processor $i + j \bmod P$, then we would achieve near-optimal performance. We found instead that if the messages that form the blocks are sent in an interleaved or random order, that the performance was better, sometimes by more than a factor of two as compared to the unsynchronized, uninterleaved, cyclic-shifts that had seemed so reasonable. For a synchronized sequence of cyclic shifts, interleaving only adds overhead without improving performance, but for message patterns about which less is known, it may often pay to interleave the order in which messages are sent.

We also found that by artificially slowing down the rate at which messages are injected into the network to exactly match the rate at which messages can be removed from the network, that we were able to achieve an additional 25% performance improvement because the network remains busy, but uncongested.

In summary, global synchronization is not only useful for correct execution of programs, it also helps improve the performance. A programmer must be careful about removing barriers because of the performance implications. Also, global synchronization in the form of a globally consistent clock makes it possible to study the performance of the system.

Massively Parallel Chess

Having designed the CM-5 to support data-parallel programming, I wanted to explore how to exploit the MIMD characteristics of the machine fully. I looked for an application that did not fit well into the data-parallel approach, and hit upon computer chess. Surprisingly, global synchronization solves problems of dynamic MIMD-style programming in addition to problems of data-parallel programming.

Computer chess provides a good testbed for understanding dynamic MIMD-style computations. The parallelism of the application derives from a dynamic expansion of a highly irregular game-tree. Thus computer chess is difficult to express as a data-parallel program. The trees being searched are orders of magnitude too large to fit into the memory of our machines, and yet serial programs can run game-tree searches depth-first with very little memory, since the search tree is at most 20 to 30 ply deep. Computer chess requires interesting global and local data structures. Computer chess is demanding enough to present engineering challenges to be solved and to provide for some interesting results, yet it is not so difficult that one cannot hope to make any progress at all. Since there is an absolute measure of performance (‘How well does the program play chess?’), there is no percentage in cheating, e.g., by reporting parallel speedups as compared to a really bad serial algorithm. In addition to those technical advantages, computer chess is also fun.

To investigate the programming issues, I engineered a parallel chess program, StarTech. StarTech is based on H. Berliner’s serial Hitech program [BE89] and runs on a Connection Machine CM-5 supercomputer. The program, running on the 512-node CM-5 at the National Center for

Supercomputing Applications at University of Illinois, tied for third place at the 1993 ACM International Computer Chess Championship, and has an informally estimated rating of 2450 USCF.[15]

The StarTech chess program is conceptually divided into two parts: The parallel game tree algorithm, which specifies *what* can be done in parallel; and the scheduler, which specifies *when* and *where* the work will actually be performed.

I found that chess places great demands on a scheduler. I found, by measuring the ideal parallelism histogram, that sometimes there is plenty of parallel work to do, and sometimes there is very little. I typically saw average available parallelism of at least several hundred, but for about a quarter of the run-time on an infinite processor machine, the available parallelism was less than 4. It is crucial that the scheduler do a good job when there is very little to do, so that the program can get back to the highly parallel parts.

To distribute work among CM-5 processors, StarTech uses a work-stealing approach, in which idle processors request work. I noticed that sometimes during a run, when the available parallelism is low, the idle processors swamp the busy processors with requests for work. This is a serious problem, since it can arbitrarily stretch out the time it takes to execute the portions of the program that have low parallelism. The swamping problem has been previously reported for work-stealing schedulers [FM87, FM93]. I provide a queueing theory explanation of how the swamping problem arises and offer a *global-throttle* mechanism to avoid the swamping problem. The global throttle organizes the computation into a series of globally synchronous phases separated by split-phase barriers. During each phase, each idle processor is allowed to make only one request to steal work. Thus the expected number of incoming requests to a busy processor is less than one. Because it uses the CM-5 control network's split-phase barriers, the busy processors continue to do useful work regardless of how long it takes for the global throttle to complete each phase.[16] Thus, the CM-5's global synchronization network is useful for dynamic MIMD programs too.

Given my scheduler, I found that two numbers, the critical path length and the total parallel work, can be used to predict the performance of StarTech. The critical path length C is the time it would take for the program to run on an infinite processor machine with no scheduling overheads. It is a lower bound to the runtime of the program. The total work W is the number of processor cycles spent doing useful work. W does not include cycles spent idle when there is not enough work to keep all the processors busy. On P processors, I define W/P to be the *linear speedup term*. The linear speedup term is also a lower bound to the runtime on P processors. Another way to think about it is to consider the program to be a dataflow graph. The critical path length is the depth of the graph, and the total work is the size of the graph. The values for C and W can be derived analytically or measured empirically. I measure the effectiveness of our scheduler by comparing it to these lower bounds. I found that the run-time on P processors of our chess program is accurately modeled as

$$T_P \approx 1.02\frac{W}{P} + 1.5C + 4.3 \text{ seconds.} \tag{1.1}$$

Except for the constant term of 4.3 seconds, this is within a factor of 2.52 of the lower bound given by the maximum of C and W/P.[17]

[15] The StarTech team has included, at various times, Hans Berliner, Mark Bromley, Roger Frye, Charles E. Leiserson, Chris McConnell, Ryan Rifkin, James Schuyler, Kurt Thearling, Richard Title, and David Waltz.

[16] Another way to solve the swamping problem is to provide separate hardware to deal with incoming requests. Such an approach could be used on the Alewife processor [ACD*91], for example. I could not add special hardware to the machine, so I had to find a software solution. Alternatively, one can use a backoff backoff strategy, such as is found in Ethernet [MB76], DIB [FM87], and PCM [HZJ94, Hal94]. The global throttle requires less tuning than does adaptive backoff, and is easy to analyze.

[17] For comparison, Brent's theorem [Bre74, Lemma 2] states that with no scheduling overhead, the runtime can be

The StarTech program uses *Jamboree* search, a parallelization of Scout search, in which at every node of the search tree, the program searches the first child to determine its value, and then tries to prove, in parallel, that all of the other children of the node are worse alternatives than the first child. This approach to parallelizing game tree search is quite natural and has been used by several other parallel chess programs [HSN89, FMM91]. While no other parallel game program uses an algorithm that is identical to StarTech's Jamboree search, I do not claim that the search algorithm is a new contribution. Instead, I view the algorithm as a testbed for testing mechanisms needed for the design of scalable, predictable, asynchronous parallel programs.

I analyzed the computational complexity of the Jamboree search algorithm, using critical path length and total work. For two special cases, best-ordered uniform trees and worst-ordered uniform trees, I used analytic methods. For best-ordered trees the critical path length is short and the amount of work performed is the same as for a good serial game-tree search. For worst-ordered trees the critical path is long, so that even on an infinite number of processors the Jamboree search achieves no speedup compared with a good serial implementation. For real chess trees I measured the critical path length and work of the program as it actually ran. I found that the quality of the move-ordering heuristics strongly affects the critical path length and the total work. I found that for tournament time controls on large machines searching real chess trees, the critical path is important but that it does not dominate the runtime. For small machines the critical path is not an issue at all.

A naive application of Jamboree search achieves work efficiencies of between 33% and 50%. The work efficiency of a parallel program on a problem is the ratio of the time for one processor to solve the problem, using the best serial code, to the total work generated by the parallel program. I found three strategies to improve the performance of StarTech, two of which exploit StarTech's global transposition table. StarTech uses a global transposition table, which memoizes results from earlier searches in order to improve move ordering for, and hence the efficiency of, later searches.[18] The first strategy for improving performance is to perform recursive iterative deepening.[19] When searching a chess position to depth k, the first thing StarTech does is to lookup the position in the global transposition table to determine if anything from a previous search has been saved. If a move for a search of depth $k-1$ or deeper is found, then StarTech uses that move as its guess for the first child. If no such move is found, then StarTech recursively searches the position to depth $k-1$ in order to find the move. By so doing, StarTech greatly improves the probability that the best move is searched first. The second strategy for improving performance is to perform *deferred-reads* on the transposition table in order to prevent more than one position from searching the same position redundantly. When a processing node starts searching a chess position, StarTech records in the global transposition table that the position is being searched. If another processor starts searching the same position, the processor waits until the first processor finishes. It is much better for the second processor to sit idle than to work on the tree, since this prevents the second processor from generating work which may then be stolen by other processors, causing an explosion of redundant work. The third strategy is to serialize Jamboree search slightly. Instead of searching one child serially and then the rest in parallel, as basic Jamboree search does, our variation sometimes searches two children serially. The precise conditions for searching two children serially are that the node be

brought down to no more than $C + W/P$. The StarTech scheduler guarantees that the memory required per processor within not much more than the memory required by the serial depth-first search. R. Blumofe and C. Leiserson studied several scheduling strategies to simultaneously achieve good time and space bounds. Some of the scheduling tradeoffs made in StarTech were strongly influenced by discussions with Blumofe and Leiserson. (See Section 5.8.)

[18] Most other programs use additional move-ordering mechanisms such as the killer table [GEC67] and the history table [MOS86]. StarTech does not currently use these additional move-ordering heuristics.

[19] Recursive iterative deepening was used in T. Truscott's unpublished checkers program in the early 1980's [Tru92], and was briefly explored for the Hitech program by H. Berliner and his students in the late 1980's [Ber93].

of Knuth-Moore type-2 [KM75], that recursive iterative search of the node had a value greater than the α parameter of the subtree, and that the search of the first child yielded a score that is less than or equal to the α parameter. This serialization improves the work efficiency of StarTech without substantially increasing the critical path length.[20]

By separating the search algorithm and the scheduler, the problems of each could be solved separately. By exploiting fast global synchronization, the problems of scheduling dynamic MIMD-style computations were simplified. I neither needed to perform arcane tuning of the scheduler nor did I need to worry about pathological search trees. Thus, I was able to focus my attention on the application, analyzing and improving the performance of the underlying search algorithm. Without the mechanisms provided by the CM-5, including global synchronization, and fast user-level messages, it would have been much more difficult to implement a competitive chess program such as StarTech.

Contributions

Global synchronization mechanisms include barriers, split-phase barriers, router-done barriers, gang scheduling, and synchronous instruction broadcast, and globally consistent clocks. What do all these have in common? Global synchronization asserts that something is true of every processor p. For example it might assert that "every processor has finished the kth step of algorithm", or "at this moment, the local clock at every processor says that the time is t".

Global synchronization provides powerful enough invariants to provide simple solutions to system problems ranging from clock distribution, diagnostics, timesharing, and spacesharing, to the correct execution of data-parallel programs, high-performance execution of dynamic MIMD-style programs, and fast bulk data transfers.

My contributions include

- Fast global synchronization is a simple, effective, efficient, solution to many system of parallel computing.

- The network architecture of the Connection Machine CM-5, a synchronized MIMD computer. The CM-5 provides three networks: a control network, a data network, and a diagnostic network. The system includes user-level network access, split-phase barriers, a router-done primitive implemented with "Kirchhoff counting", a split data network to allow commonly used protocols to be implemented without deadlock or bookkeeping, an "all-fall-down" mechanism that quickly empties the routing network for timesharing, and a parallel diagnostics strategy. The system efficiently supports data-parallel programming and dynamic MIMD-style programming.

- Three strategies for obtaining good performance from a data network: periodic barriers, injection reordering, and bandwidth matching.

- StarTech, a competitive parallel chess program that runs on the CM-5. StarTech employs a work-stealing scheduler, which uses the CM-5 control network to throttle processors that are looking for work. The StarTech scheduler achieves good performance relative to optimal scheduling. StarTech uses Jamboree search, a parallel game tree search algorithm. I analyze the performance of Jamboree search on special case game trees, and measure the performance

[20]Charles E. Leiserson and I together designed the serialization heuristic for Jamboree search.

on real chess trees. I systematically employ the critical path length and total work to understand the performance of StarTech, which is a dynamic MIMD-style program. I demonstrate a heuristic that improves the work efficiency of the Jamboree algorithm on real chess trees. StarTech uses a global transposition table, the performance of which is improved by using recursive iterative deepening and deferred lookups. StarTech uses several active message protocols that may be useful in a wider context to avoid dangling references and reduce message traffic for atomic access to global data-structures.

A Road Map

This dissertation is organized into three, mostly independent, parts.

- The first part (Chapter 2) describes the network architecture of the CM-5.
- The second part (Chapter 3) describes how to get good performance from the CM-5 data network, especially for the kinds of message traffic that show up in data-parallel programs.
- The third part (Chapters 4–6) describes the StarTech massively parallel chess program.

While the three parts of the thesis together provide evidence that fast global synchronization is useful for MIMD computation, the ideas described in each part stand alone, and the parts can be read independently. Many of the mechanisms of Part I can be employed to solve system-level problems in other kinds of parallel machines. Such problems include deadlock avoidance, timesharing, and detecting the termination of a computation. The mechanisms of Part II can be used to improve the performance of data transfers on almost any machine, although machines without fast global synchronization are at a disadvantage compared to synchronized MIMD machines. The results of Part III include an algorithm, with analysis, for parallel game tree search, and mechanisms for scheduling work on parallel machines.

Chapter by chapter, this dissertation is organized as follows:

- Chapter 2 describes the CM-5, a specific instance of the synchronized MIMD architecture. Beginning with an explanation of how data-parallel code can be compiled for and executed on a synchronized MIMD machine, the chapter proceeds to explain the network architecture of the CM-5. The CM-5 includes three networks, a data network, a control network, and a diagnostics network. The CM-5 includes mechanisms to support time-sharing, and space-sharing; to detect the completion of computations that use the data network; and to help the user program the data network without deadlocking. The diagnostics network can quickly test a large CM-5 in parallel.
- Chapter 3 continues the study of the CM-5 by examining how to get good performance from the CM-5 data network. The chapter starts with an explanation of the problem, that removing barriers can sometimes slow down a computation, and reviews some of the issues that arise when measuring performance on the CM-5. The chapter then shows how and why programmers can improve the performance of their data movement code by more than a factor of three by selectively using global barriers; by limiting the rate at which messages are injected into the network; and by managing the order in which they are injected.
- Chapter 4 begins the second part of this dissertation by explaining how the StarTech chess program works. The chapter describes the Jamboree search algorithm, starting with a review of game tree search, and explains how Jamboree search is related to the serial α-β and Scout

search algorithms. The chapter then describes the StarTech work-stealing scheduler, which employs global synchronization to help guarantee good performance. The Jamboree algorithm employs speculative expansion of the tree, and sometimes the algorithm discovers that a partially expanded subtree is no longer needed. Startech employs a simple active-message protocol that avoids dangling references to frames that are aborted in mid-computation.

- Chapter 5 studies the performance of the StarTech program, beginning with a discussion of Brent's theorem. The chapter presents a study of the Jamboree algorithm, analyzing two special cases: best-ordered uniform game trees and worst-ordered uniform game trees. Expressions are produced for the critical path length, the parallel work, and the work efficiency of Jamboree search on such trees. An empirical study of the Jamboree algorithm is presented, focusing on the critical path length and parallel work of the algorithm when search real chess trees. Having examined the Jamboree algorithm, the chapter moves on to study the StarTech scheduler, starting with a demonstration that our parallel chess program places great demands on the scheduler. The scheduler's performance is then analyzed in relation to critical path length and linear-speedup lower bounds to performance, showing that Equation 1.1 accurately models the performance of StarTech. In particular, one can measure the performance of the program on a small machine, and predict its performance on a large machine. A study of the swamping problem yields justification for the global throttling strategy of the StarTech scheduler. Using a simple experimental setup, the swamping problem is demonstrated, and an analytic queueing model is developed that provides a good match to the empirical measurements. The chapter concludes with an analysis of the space-time tradeoffs associated with fixing each position of the chess tree on a particular processor rather than allowing the work to migrate from one processor to another.

- Chapter 6 explains how I used the critical path length and parallel work measurements to improve the performance of StarTech. Chapter 6 describes our global transposition table, and explains how recursive iterative deepening and deferred-reads are implemented and what are the performance implications of those mechanisms. I analyze where the extra work is coming from that leads to StarTech's modest work inefficiency. Using that analysis, I construct a modification to the basic Jamboree search that sometimes serializes the search. That modification, when used, has effect of significantly decreasing the total parallel work while increasing the critical path length only slightly. The chapter concludes with a study of how the processor-cycles are spent by StarTech, with an eye to understanding how to improve the performance of the program in the future.

- Chapter 7 concludes by reviewing the relationship of this work to previous work, and discussing the merits of the mechanisms that I describe, both with respect to today's technology and to that of the future.

Chapter 2

The Network Architecture of the Connection Machine CM-5[1]

In the design of a parallel computer, the engineering principle of *economy of mechanism* suggests that the machine should employ only a single communication network to convey information among the processors in the system. Indeed, many parallel computers contain only a single network: typically, a hypercube or a mesh. The Connection Machine Model CM-5 Supercomputer has three networks, however, and none is a hypercube or a mesh. This chapter describes the architecture of each of these three networks and the rationale behind them.

The CM-5 is a *synchronized MIMD* machine, which combines the best aspects of SIMD (single instruction path, multiple data path) and MIMD (multiple instruction path, multiple data path) machines [Fly66]. Each processor in the CM-5 executes its own instructions, providing the flexibility of a typical MIMD machine. And, like many MIMD machines, the CM-5 is a distributed memory machine (as opposed to shared memory machine [DT90, GGK*83]) in which processors communicate among themselves by sending messages [Sei85, SAD*86] through the *data network* of the machine. A deficiency of typical MIMD machines, especially as compared with their SIMD cousins, however, is that they provide little or no support for coordinating and synchronizing sets of processors. To offset this deficiency, the CM-5 also contains a *control network*, which makes synchronization and multiparty communication primitives competitive with comparable functions on SIMD machines. These primitives include the fast broadcasting of data, barrier synchronization [Jor78, TY86, DGN*86], and parallel prefix (scan) operations [Ble90].

Figure 2-1 shows a diagram of the the CM-5 organization. The machine contains between 32 and 16,384 *processing nodes*, each of which contains a 32-megahertz SPARC processor, 32 megabytes of memory, and a 128-megaflops vector-processing unit capable of processing 64-bit floating-point and integer numbers. System administration tasks and serial user tasks are executed by a collection of *control processors*, which are Sun Microsystems workstation computers. There are from 1 to several tens of control processors in a CM-5, each configured with memory and disk according to the customer's preference. Input and output is provided via high-bandwidth *I/O interfaces* to graphics devices, mass secondary storage, and high-performance networks. Additional low-speed I/O is provided by Ethernet connections to the control processors. The largest machine, configured with up to 16,384 processing nodes, occupies a space of approximately 30 meters by 30 meters, and is capable of over a teraflops (10^{12} floating-point operations per second).

[1]The research in this chapter represents joint work with Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Margaret A. St. Pierre, David S. Wells, Monica

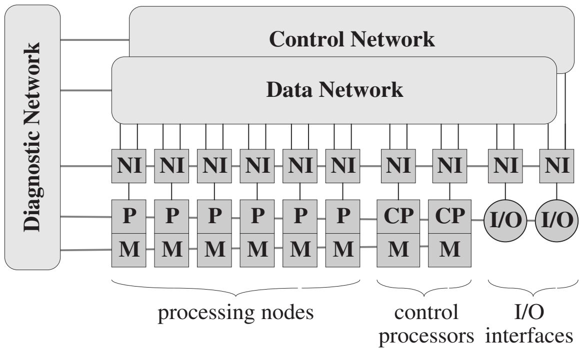


Figure 2-1: The organization of the Connection Machine CM-5. The machine has three networks: a data network, a control network, and a diagnostic network. The data and control networks are connected to processing nodes, control processors, and I/O channels via a network interface.

The processing nodes, control processors, and I/O interfaces are interconnected by three networks: the data network, the control network, and a diagnostic network. The data network provides high-performance point-to-point data communications between system components. The control network provides cooperative operations, including broadcast, synchronization, and *scans* (parallel prefix and suffix). It also provides system management operations, such as error reporting. The diagnostic network allows "back-door" access to all system hardware to test system integrity and to detect and isolate errors.

The system operates as one or more user *partitions*. Each partition consists of a control processor, a collection of processing nodes, and dedicated portions of the data and control networks. Access to system functions is classified as either *privileged* or *nonprivileged*. All nonprivileged system functions, including access to the data and control networks, can be executed directly by user code without system calls. Consequently, network communication within a user task occurs without operating system overhead. Access to the diagnostics network, to shared system resources (such as I/O), and to other partitions is privileged and must be accomplished via system calls. Protection and addressing mechanisms ensure that no user can interfere with the function or performance of another user in another partition. If the system administrator so desires, a single partition can be timeshared among a group of users, where each user gets a fair portion of the available time and cannot otherwise be interfered with by any other user.

This chapter describes the CM-5 synchronized MIMD hardware and how to use it to run data parallel programs. The rest of this chapter then focuses on the details of the network architecture of the CM-5, and is organized as follows. Section 2.1 describes the network interface which provides the user's view of the data and control networks. Section 2.2 then describes the data network, Section 2.3 describes the control network, and the diagnostic network is described in Section 2.4. Section 2.5 discusses how global synchronization helped solve many system problems in the CM-5. The chapter closes with Section 2.6, which gives a short history of our development project.

Further details about the CM-5 system can be found in the CM-5 Technical Summary [TMC91]. The reader should be aware that the performance specifications quoted in this chapter apply only to the initial release of the CM-5 system. Because of our ability to reengineer pieces of the system easily, these numbers represent only a snapshot of an evolving implementation of the architecture.

C. Wong, Shaw-Wen Yang, and Robert Zak. Much of the work in this chapter was originally reported in [LAD*92].

The machine has recently been revised to include faster processors and data networks.

2.1 The CM-5 Network Interface

Early on in the design process, we, the CM-5 development team at Thinking Machines Corporation, decided to specify an interface between the processing nodes and the networks that isolates each from the details of the other. This interface provides three features. First, the interface gives the processors a simple and uniform view of the networks (and the networks get a simple and uniform view of the processors). Second, the interface provides support for time-sharing, space-sharing, and mapping out of failed components. Third, the interface provides a contract for the implementors which decouples the design decisions made for the networks from those of the processors.

The processor's view of the interface is as a collection of memory-mapped registers. By writing to or reading from fixed physical memory addresses, data is transferred to or from the networks, and the interface interprets the particular address as a command.

A memory mapped interface allows us to use many of the memory-oriented mechanisms found in off-the-shelf processors to deal with network interface issues. To access the network, a user or compiler reads from or writes to locations in memory. We regarded the prospect of executing a system supervisor call for every communication as unacceptable, in part because we wished to support the fine-grain communication needs of data-parallel computation. A memory-mapped interface allows the operating system to deny users access to certain network operations by placing the corresponding memory-mapped registers on protected pages of the processor's address space. The processor's memory management unit enforces protection without any additional hardware.

The interface is broadly organized as a collection of memory-mapped FIFO's. Each FIFO is either an *outgoing* FIFO to provide data to a network, or an *incoming* FIFO to retrieve data from a network. Status information can be accessed through memory-mapped registers. For example, to send a message over a network, a processor pushes the data into an outgoing FIFO by writing to a suitable memory address. When a message arrives at a processor, the event is signaled by interrupting the processor, or alternatively, the processor can poll a memory-mapped status bit. The data in the message can then be retrieved by reading from the appropriate incoming FIFO. This paradigm is identical for both the data and control networks.

The network interface provides the mechanisms needed to allow context switching of user tasks. Each user partition in the CM-5 system can run either batch jobs or a timesharing system. When a user is swapped out during timesharing, the processors must save the computation state. Some of this state information is retrieved from the network interface, and the rest is garnered from the networks. The context-switching mechanism also supports automatic checkpointing of user tasks.

The interface provides processor-address mapping so that the user sees a 0-based contiguous address space for the processor numbers within a partition. Each processor can be named by its *physical* address or by its *relative* address within the partition. A physical address is the actual network address as interpreted by the hardware inside the networks. A relative address gives the index of a processor relative to the start of a user partition, where failed processors are mapped out. All processor addresses in user code are relative addresses. To specify physical addresses requires supervisor privileges. Relative addresses are bounds checked, so that user code cannot specify addresses outside its partition.

The user's view of the networks is independent of a network's topology. Users cannot directly program the wires of the networks, as they could on our previous machine, the CM-2. The reason is simple: the wires might not be there! Because the CM-5 is designed to be resilient in the presence of faults, we cannot allow the user to rely on a specific network topology. One might

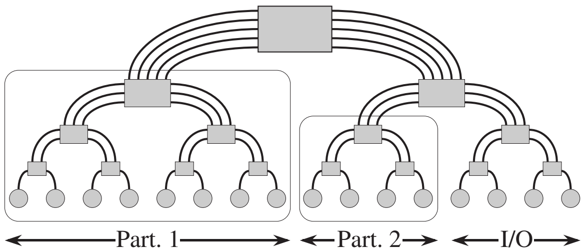


Figure 2-2: A binary fat-tree. Processors are located at the leaves, and the internal nodes are switches. Unlike an ordinary binary tree, the channel capacities of a fat-tree increase as we ascend from leaves to root. The hierarchical nature of a fat-tree can be exploited to give each user partition a dedicated subnetwork which cannot be interfered with by any other partition's message traffic. The CM-5 data network uses a 4-ary tree instead of a binary tree.

think topology independence would hurt network performance, but we found this presumption to be less true than we initially imagined. Because we did not provide the user with access to the wires of the network, we were able to apply more resources to generic network capabilities. A further advantage of topology independence is that the network technology becomes decoupled from processor technology. Any future network enhancements are independent of user code and processor organization.

An important ramification of the decoupling of the processors from the networks is that the networks must assume full responsibility for performing their functions. The data network, for example, does not rely on the processors to guarantee end-to-end delivery. The processors assume that delivery is reliable. Nondelivery implies a broken system, since there is no protocol for retransmission. By guaranteeing delivery, additional error-detection circuitry must be incorporated into the network design, which slightly reduces its performance, but since the processor does not need to deal with possible network failures, the overall performance as seen by a user is much better.

The CM-5 network interface is implemented in large measure by a single 1-micron standard-cell CMOS chip, with custom macro cells to provide high-performance circuits where needed. The interface chip is clocked by both the 32-megahertz processor clock and the 40-megahertz networks clock. Asynchronous arbiters synchronize the processor side of the interface with the network side.

Choosing to build a separate network interface allowed the processor designers to do their jobs and the network designers to do theirs with a minimum of interference. As a measure of its success in decoupling the networks from the processor organization, the same interface chip is used to interface the network to I/O channels, of which there are many types, including CMIO, VME, FDDI, and HIPPI.

2.2 The CM-5 Data Network

The basic architecture of the CM-5 data network is a *fat-tree* [GL89, Lei85]. Figure 2-2 shows a binary fat-tree. Unlike a computer scientist's traditional notion of a tree, a fat-tree is more like a real tree in that it gets thicker further from the leaves. Processing nodes, control processors, and I/O channels are located at the leaves of the fat-tree. (For convenience, we shall refer to all of these network addresses simply as processors.)

A user partition corresponds to a subtree in the network. Messages local to a given partition are

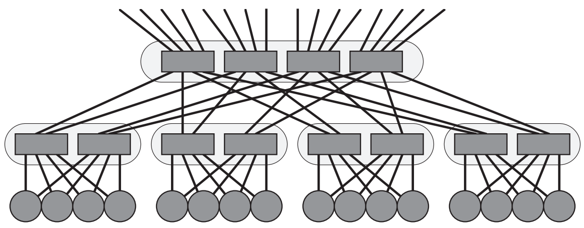


Figure 2-3: The interconnection pattern of the CM-5 data network. The network is a 4-ary fat-tree in which each internal node is made up of several router chips. Each router chip is connected to 4 child chips and either 2 or 4 parent chips.

routed within the partition's subtree, thereby requiring no bandwidth higher in the tree. Access to shared system resources, such as I/O, is accomplished through the part of the fat-tree not devoted to any partition. Thus, message traffic within a partition, between a partition and an I/O device, or between I/O devices does not affect traffic within any other partitions. Moreover, since I/O channels can be addressed just like processing nodes, the data network becomes a true "system bus" in which all system components have a unique physical address in a single, uniform name-space.

Of critical importance to the performance of a fat-tree routing network is the communication bandwidth between nodes of the fat-tree. Most networks that have been proposed for parallel processing, such as meshes and hypercubes, are inflexible when it comes to adapting their topologies to the arbitrary bandwidths provided by packaging technology. The bandwidths between nodes in a fat-tree, however, are not constrained to follow a prescribed mathematical formula. A fat-tree can be adapted to effectively utilize whatever bandwidths make engineering sense in terms of cost and performance. No matter how the bandwidths of the fat-tree are chosen, provably effective routing algorithms exist [GL89, LMR88] to route messages near-optimally. The underlying architecture and mechanism for addressing is not affected by communication bandwidths: to route a message from one processor to another, the message is sent up the tree to the least common ancestor of the two processors, and then down to the destination.

Because of various implementation trade-offs—including the number of pins per chip, the number of wires per cable, and the maximum cable length—we designed the CM-5 data network using a 4-ary fat-tree, rather than a binary fat-tree. Figure 2-3 shows the interconnection pattern. The network is composed of router chips, each with 4 *child* connections and either 2 or 4 *parent* connections. Each connection provides a link to another chip with a raw bandwidth of 20 megabytes/second in each direction. (Some of this bandwidth is devoted to addressing, tags, error checking, etc.) By selecting at each level of the tree whether 2 or 4 parent links are used, the bandwidths between nodes in the fat-tree can be adjusted. Flow control is provided on every link.

Based on technology, packaging, and cost considerations, the CM-5 bandwidths were chosen as follows. Each processor has 2 connections to the data network, corresponding to a raw bandwidth of 40 megabytes/second in and out of each processing node. In the first two levels, each router chip uses only 2 parent connections to the next higher level, yielding an aggregate bandwidth of 160 megabytes/second out of a subtree with 16 processing nodes. All router chips higher than the second level use all 4 parent connections, which, for example, yields an aggregate bandwidth of 10 gigabytes/second, in each direction, from one half of a 2K-node system to the other. The bandwidth continues to scale linearly up to 16,384 nodes, the largest machine that Thinking Machines can currently build. (The architecture itself scales to over one million nodes.) In larger machines, transmission-line techniques are used to pipeline bits across long wires, thereby over-coming the bandwidth limitation that would otherwise be imposed by wire latency. The machine is designed so that network bandwidth can be enhanced in future product revisions without affecting the architecture.

The network design provides many comparable paths for a message to take from a source processor to a destination processor. As it goes up the tree, a message may have several choices as to which parent connection to take. This decision is resolved by pseudorandomly selecting from among those links that are unobstructed by other messages. After the message has attained the height of the least common ancestor of the source and destination processors, it takes the single available path of links from that chip down to its destination. The pseudorandom choice at each level balances the load on the network and avoids undue congestion caused by pathological message sets. (Many naive algorithms for routing on mesh and hypercubic networks suffer from having specific message patterns that do not perform well, and the user is left to program around them.) The CM-5 data network routes all message sets nearly as well as the chosen bandwidths allow.

A consequence of the automatic load balancing within the data network is that users can program the network in a straightforward manner and obtain high performance. Moreover, an accurate estimate of the performance of routing a set of messages through the network can be predicted by using a relatively simple model [LM88]. One determines the load of messages passing through each arm of the fat-tree and divides this value by the available bandwidth. The worst-case such ratio, over all arms of the fat-tree, provides the estimate.

On random permutations, each processor can provide data into, and out of, the network at a rate in excess of 4 megabytes/second. When the communication pattern is more local, such as nearest neighbor within a regular or irregular two- or three-dimensional grid, bandwidths of 15 megabytes/second per processor are achievable. The network latency ranges between 3 and 7 microseconds, depending on the size of the machine. All of these empirical values include the time required for processors to execute the instructions needed to put messages into and take messages out of the network.

The data network is currently implemented from 1-micron standard-cell CMOS chips, with custom macro cells to provide high-performance circuits where needed. Each chip has an 8-bit-wide bidirectional link (4 bits of data in each direction) to each of its 4 child chips lower in the fat-tree, and 4 8-bit-wide bidirectional links to its parent chips higher in the fat-tree. The data-network chip can be viewed as a crossbar connecting the 8 input ports to the 8 output ports, but certain input/output connections are impossible due to the nature of the routing algorithm. For example, we never route a message from one parent port to another. When a message is blocked from its desired output port, it is buffered. Flow control information is passed in the reverse direction of message traffic to prevent buffer overflow. When multiple messages compete for the same output port, the arbitration is fair and prevents any link from being starved. We designed only one chip to do message routing, and we use the same chip for communication between chips on the same circuit board as between chips that are in different cabinets.

Interchip data is sent on differential pairs of wires, which increases the pin count of the chips, but which provides outstanding noise immunity and reduces overall power requirements.[2] We rejected using separate transceivers at the packaging boundaries, because it would have increased power consumption, board real estate, and the number of different chips we would have needed to design, debug, test, stock, etc. The diagnostics can independently test each conductor of each differential signal, because differential signals are so immune to noise that they sometimes work even with broken wires.

[2]Our differential drivers and receivers are relatively straightforward. A more complex design that includes self-terminating transceivers is described by T. Knight and A. Krymm in [KK88].

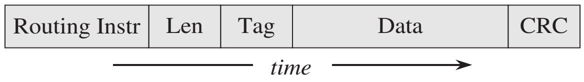


Figure 2-4: The format of messages in the data network. Each message contains routing instructions, a length field that indicates how many data words are in the message, a tag field that indexes an interrupt vector in the processor, data words, and a cyclic redundancy check.

The first 2 levels of the data network are routed through backplanes. The wires on higher levels are run through cables, which can be either 9 or 26 feet in length. The longer cables maintain multiple bits in transit. The wires in cables are coated with expanded Teflon, which has a very low dielectric constant. The cables reliably carry signals in excess of 90 percent of the speed of light.

The data network chips are clocked synchronously by a 40-megahertz clock. The clock is distributed with very low skew—even for the biggest machines—by locally generating individual clocks and adjusting their phases to be synchronous with a centrally broadcast clocking signal.[3] The local generation of clocks also protects the machine from a catastrophic single point of failure; without the redundancy, a single central clock could fail, causing the entire machine's power consumption to drop to nearly zero in under one microsecond, possibly damaging the power distribution system.

Messages routed by the data network are formatted as shown in Figure 2-4. The beginning of the message contains routing instructions that tell how high the message is to go in the tree and then the path it is to follow downward after it reaches its zenith. The routing instructions are chip-relative instructions that allow each chip to make a simple, local decision on how to route the message. Following the routing instructions is a field that indicates the length of the data in 32-bit words. Currently, the CM-5 network interface allows between 1 and 5 words. Longer messages must be broken into smaller pieces. Following the length field is a 4-bit tag field that can be used to distinguish among various kinds of messages in the system. The network interface interprets some of these tags as system messages, and the rest are available to the user. When a message arrives at a processor, the tag indexes a 16-bit mask register in the network interface, and if the corresponding mask bit is 1, the processor is interrupted. After the tag comes the data itself, and then a field that provides an integrity check of the message using a cyclic redundancy code (CRC).

Because we desired to build very large machines, we deemed it essential to monitor and verify the data network dynamically, because the chances of a component failure increase with the size of the system. Message integrity is checked on every link and through every switch. If a message is found to be corrupted, an error is signaled. Messages snake their way through the switches in a manner similar to cut-through [KK79] or worm-hole [Dal87, DS87] routing, and so by the time that a data-network chip has detected an error, the head of the message may have traveled far away. To avoid an avalanche of errors, the complement of a proper CRC is appended to the message. Any chip that discovers the complement of a proper CRC signals a secondary error. Thus, a typical error causes one chip to signal a primary error with a trail of chips reporting secondary errors, although there is some positive probability that a primary error is reported as a secondary error. Diagnostic programs can easily isolate the faulty chip or link based on this information, which is accessible through the diagnostic network. Lost and replicated messages can be detected by counters on each chip and in the network interfaces that maintain the number of messages that pass on each

[3]More information about our clocking technique can be found in [HAK*]. A related clocking strategy, in which skew is compensated for by adjusting the phase of each data signal, is described by P. Bassett *et al.* in [BGR86].

link. Using a variation on Kirchoff's current law, the number of messages entering any region of the network, including the entire network or a single chip, must eventually equal the number of messages leaving the region. This condition is checked for the entire data network by the control network (see Section 2.3).

Once a faulty processor node, network chip, or interconnection link has been identified, the fault is mapped out of the system and quarantined. The network interface allows for mapping faulty processing nodes out of the network address space. The rest of the system ignores all signals from the mapped-out portion, thereby allowing the system to remain functional while servicing and testing, or even powering down, the mapped-out portion.

When a chip or link in the data network fails, there are two mechanisms to map around the fault. Either the network can be configured to route messages away from the failure, or processing nodes that might use the chip or link can be mapped out. By picking the better of the two alternatives, the system can guarantee either that at most 6 percent of the network is lost or that at most $1/64$ of the processing nodes are mapped out.

The network has a contract with processors that guarantees all messages are delivered:

> **contract:**
>
> *The data network promises to eventually accept and deliver all messages injected into the network by the processors as long as the processors promise to eventually eject all messages from the network when they are delivered to the processors.*

The data network is acyclic from inputs to outputs, which precludes deadlock from occurring if this contract is obeyed. To send a message, a processor writes the destination processor address and data to be sent to a memory-mapped outgoing FIFO in its network interface. The processor then checks whether the message was accepted by the network. If not, which may occur because flow control information indicates that the network has not removed enough of a previous message from the outgoing FIFO, the processor can try again later. The processor may not block or spin when attempting to put a message into the network, however, because that would violate the contract. Instead, the processor must attempt to receive any messages that have arrived. In the current implementation, the processor is involved in all transactions with the network.

Although the simple contract above can implement the sending of data through the network in a deadlock-free manner, it is not strong enough to allow some communication protocols to be implemented straightforwardly. Consider, for example, the *fetch-deadlock problem*: each processor wishes to fetch a value from another processor, and the processors have finite buffer space. The message traffic for a protocol that solves this problem corresponds to a round trip in the network: a request from one processor to another, followed by a response from the other to the one. In this scenario, one processor may receive requests for data from many processors, but unfortunately, be unable to send responses because its outgoing FIFO to the data network is busy. The outgoing FIFO will eventually free, according to the contract, but only if the processor continues to accept delivery of messages from the network. With finite buffer space, however, there is a limit to how many requests it can handle. When it runs out of buffer space, the processor will be forced to refuse delivery, thereby breaking the contract, and deadlock may result.

With buffer space proportional to the number of processors in the system, it is possible to construct a "round-trip" protocol that solves the fetch-deadlock problem. The key idea is to program a reservation mechanism [Kle78] that ensures that at most a bounded number of messages are outstanding between any two processors at any time. A processor X does not attempt to send a message to another processor Y until Y informs X that it has room to handle the message. This protocol, which has been implemented on some parallel computing systems, including the CM-5, requires a substantial software overhead for bookkeeping.

The CM-5, however, provides another way to solve the fetch-deadlock problem in a simple fashion requiring no bookkeeping and only constant buffer space. Each processor has 2 outgoing and 2 incoming FIFO's in its interface to the data network: a *left* port and a *right* port. The topology of the network is such that all links reachable from the left port are unreachable from the right port and vice versa. Thus, the data network is really two independent, interleaved networks. To implement the round-trip protocol, requests can be sent on the left side of the network, and responses returned on the right side. If a processor cannot send a response on the right side and his constant-size buffer is full, he stops receiving on the left side. Since any processor requesting data has a place to put it, however, the processors can satisfy the contract on the right side, and the responses will eventually clear out. Because the responses on the right side will eventually clear out, a processor can always eventually accept every request that arrives on the left side, and thus the processors satisfy the contract on the left side. Consequently, deadlock cannot occur.

In fact, deadlock cannot occur even if responses are sent on both sides of the data network, as long as requests are sent on one side only. The data network requires no more than two sides, even when there are many intermediate destinations, because such a communication pattern can be broken into a collection of round trips.

The CM-5 programming systems (CM-Fortran, C*, and *Lisp) never allow a user to deadlock, because they implement deadlock-free protocols for communication. Deadlock can occur, however, if a programmer chooses to program the individual processing nodes directly. All he need do is break the contract that the processing nodes have with the data network: he writes code that sends messages but never attempts to receive them. This danger may seem quite alarming, but it is no more alarming than the danger that a user writes an infinite loop. On the CM-5, the user can send and receive messages without executing a system call, as is required on many other systems. By giving the user direct access to the network, the user can in some circumstances obtain greater efficiency than he could obtain with the communication routines available in the standard system libraries. If he does deadlock himself, or write an infinite loop, he does not affect any other user.

Each user partition in the CM-5 system is capable of being run in either a batch or a timesharing mode. The requirement for timesharing raises the issue of what should be done with messages that are in transit in the routing network when a user's timeslice has expired and another user must be given access to the partition. The system cannot afford to wait until the user completes his communication, since the communication may not terminate for a very long time, and, in fact, it may never complete if the user has deadlocked himself.

We considered several solutions to the problem of swapping users. For example, we considered entering a special routine that would pull messages out of the router and discard them. This solution was considered too expensive, because the user would be constantly forced to checkpoint the computation so that the discarded messages could be reconstructed. Moreover, if the user fills the network with messages that are all addressed to the same processing node, then the time to empty the router would be proportional to the machine size, which was deemed unacceptably long.

This problem of swapping users is solved in the CM-5 by putting the data network into *all-fall-down* mode. Instead of trying to route messages to their destinations, when a data-network chip is in all-fall-down mode, each message is routed downward according to a fixed permutation that has been preprogrammed by the system and which ensures that all-fall-down messages are distributed evenly among the processing nodes. In the worst case, each node receives only a small number of misdirected messages, even if all messages were headed for the same destination processor. The all-fall-down messages can then be saved in memory with the user's state. When the user's task is resumed, the system resends them to their true destinations. Even if a timeshared user deadlocks, this context-switching mechanism precludes him from unduly affecting the other users who are sharing his partition.

During our design of the all-fall-down mechanism, the problem arose of how to set all chips of a partition into this mode. We considered engineering a mechanism in which all chips were put into all-fall-down mode simultaneously, but even so, we considered it a lot of detailed engineering work to guarantee that messages between chips when all-fall-down was initiated would be handled properly. Instead, we adopted a simple protocol to allow chips in the data network to be put into all-fall-down mode in any order. The basic idea is that all-fall-down messages are marked as such. When a chip sees an all-fall-down message, it routes it downward according to the preprogrammed permutation, even if the chip is not in all-fall-down mode and is routing other messages in a normal fashion. Thus, once a message starts falling, it keeps falling and is never interpreted by any chip as anything but an all-fall-down message.

In summary, the CM-5 data network provides fast point-to-point communication of data, but as importantly, it provides flexible solutions to system problems.

2.3 The CM-5 Control Network

There are two general classes of operations on the control network: broadcasting and combining. Separate FIFO's in the network interface correspond to each type of control-network function. A processor pushes a message into one of the outgoing FIFO's, and shortly after all processors have pushed messages, the result becomes available to all processors as messages in their respective incoming FIFO's.

Every operation on the control network potentially involves every processing node. Broadcast messages from the control processor are replicated at nodes in the tree and distributed to the subtrees. Other operations, such as scans (parallel prefix), require input from all processors and provide output to all processors. The control network is pipelined, so that several messages can be sent before any are received. To provide further flexibility, each processing node can set up the network interface to abstain from certain control-network operations. These operations complete as if the abstaining processors had provided "identity" data, but without making them waste processing cycles. Overall, the control network is designed to support cooperative functions that require little bisection bandwidth, and hence, which can be implemented efficiently on a simple tree.

Broadcasting

A processor may broadcast a message through the control network to all other processors in its partition. The control network supports four kinds of broadcasting: user broadcast, supervisor broadcast, interrupt broadcast, and utility broadcast. User and supervisor broadcasts are essentially identical, except that supervisor broadcasts are privileged operations. These broadcast operations can be used to download code and to distribute data. An interrupt broadcast is a privileged operation that causes every processor to receive an interrupt. Interrupt broadcasts provide the ability to "grab the attention" of all processors in the user partition, which is especially useful for implementing operating system functions, such as swapping timeshared users. The utility broadcast is used by the operating system to configure partitions and to perform other sorts of system operations.

Only one processor may broadcast at a time, but broadcasts are pipelined so that the broadcasting processor can fully utilize the broadcast bandwidth of the network. If, while one processor is broadcasting, another processor sends a broadcast message, the control network signals an error when the competing messages collide. The number of simultaneous pipelined broadcasts supported by the control network depends upon the height of the network partition. The current implementation of the CM-5 provides the user with up to 8 words in a broadcast and the supervisor with up to 4 words.

Combining

The control network supports four different types of combining operations: reduction, forward scan (parallel prefix), backward scan (parallel suffix), and router done. Moreover, the network interface chip is capable of masking out processors that do not wish to participate in a control-network operation, so that operations can be performed only on a subset of the processors in a partition. Only one combining operation can be initiated at a time, but the network is pipelined, which allows several operations to be initiated rapidly in sequence.

A reduction operation combines values provided by all (participating) processors according to a user-supplied operator and delivers a copy of the result to all processors. Messages are combined with one of five operators on 32-bit words: bitwise logical OR, bitwise logical XOR, signed maximum (which also works for IEEE floating-point numbers), signed addition, and unsigned addition. (The two addition operators differ in how overflow is reported.) Reductions over other commonly occurring operators (such as bitwise logical AND) can be easily synthesized from these and local processor operations. The control network also supports reductions on values larger than 32 bits by a sequence of 32-bit reductions, each of which saves residual data, such as a carry in the case of addition, which is input to the next reduction in the sequence. Since the control network is pipelined, the latency for a multiple-word reduction operation is not unduly affected.

A forward scan operation delivers to the ith processor the result of applying one of the five reduction operators to the values in the preceding $i - 1$ processors (in the linear order given by data network address). For example, a forward scan of the vector $\langle 3, 2, 0, 4, 2, 6, 5, 8 \rangle$ with the operator $+$ yields the vector $\langle 0, 3, 5, 5, 9, 11, 17, 22 \rangle$. A backward scan provides similar functionality in the reverse direction. Scans can be segmented: if a "segment start" bit in the network interface is set, the scan starts over at that point. Backward scans are also supported. All basic scan operations use 1-word (32-bit) inputs, but multiple-word scans are supported by a sequence of 1-word scans in a manner similar to multiple-word reductions. An excellent discussion of scans can be found in [Ble90].

Early on in the design of the CM-5, we decided to support scans in hardware. Our experience with the CM-2 showed that many high-performance data-parallel algorithms—including both combinatorial and numerical algorithms—make extensive use of scans. The operations that were selected (OR, XOR, etc.) reflect a compromise between making the hardware fast and simple and providing sufficient building blocks out of which other operations could be constructed. For instance, OR can be used to implement AND (DeMorgan's law), so there is no need to implement both. As a more sophisticated example, segmented reductions, which are not provided directly by the hardware, can be implemented by using two segmented scans, one forward and one backward. Since the control network is pipelined, the overhead of doing both is minimal.

The router-done operation is a specialized reduction that lets the processors know when communications involving the data network are complete. In the data-parallel programming model, this operation is often required so that processors know when it is safe to proceed to the next data-parallel operation.

The basic idea behind the implementation of router-done is "Kirchhoff's current law." When all processors have completed sending their messages and the number of messages that entered the data network equals the number that have left, the routing cycle is complete. The network interfaces keep track of the number of messages that enter and leave the data network. After a processor has completed sending all its messages, it pushes a message into the outgoing router-done FIFO. When all processors have sent messages into their outgoing FIFO's, the control network continually monitors the difference between the total number of messages put into the data network and the number removed from the data network. When this number becomes zero, each processor receives a message in its incoming router-done FIFO informing it that the data network is done routing messages.

There are several other software and hardware approaches to computing a router-done barrier. A software solution involves sending an acknowledgment for every delivered message. Each processor enters a normal, processor-only, barrier only when it has received acknowledgments for all the messages it sent. Such a scheme is expensive, since it doubles the network traffic. The Monsoon dataflow processor employs a network that provides a hardware acknowledgment for every message [PC90].[4] The Connection Machine CM-1 [Hil85] uses a global-or circuit wired into all the network chips to determine when the network is empty. For the CM-5, we considered a strategy that uses special tokens to "sweep" the network clean.

Using this "Kirchhoff" method has the additional benefit that if a hardware error causes messages to be lost or created, the error can be detected and signaled, either by a failure of the router-done operation to complete on the one hand or by the unexpected arrival of a message after the router-done operation has completed on the other.

The CM-5 control network also supports one synchronous OR operation and two identical asynchronous OR operations that can operate in parallel with other network operations, and have separate FIFO's in the network interface. The synchronous OR is similar to an OR reduction, except that a processor's input and output each consist of only a single bit. Each asynchronous OR operates continuously without waiting for all processors to participate. Processors are free to change their inputs at any time and sample the output. The asynchronous OR can be used for signaling conditions and exceptions. The transition of an asynchronous OR from 0 to 1 can be used to signal an interrupt. One of the two asynchronous OR's is privileged, and the other is nonprivileged.

The synchronous OR or any of the various combining operations can be used to implement *split-phase* barrier synchronization [TMC88]. (In independent work [Gup89], this type of synchronization has been called a *fuzzy barrier*.) In a split-phase barrier, the barrier is a region of code with an entry and an exit. (If the region is empty, an ordinary barrier results.) When a processor enters the split-phase barrier, it pushes an input message into an appropriate outgoing FIFO. Shortly after all other processors have pushed their messages, they all receive messages from the corresponding incoming FIFO, and each can infer that all have entered the barrier. The advantage of a split-phase barrier over an ordinary barrier is that the processor can execute code while waiting for the barrier to complete. Thus, just as the instruction following a delayed branch in a RISC architecture can compensate for the latency of the branch, the code between barrier entry and exit can compensate for the latency of synchronization. The router-done operation couples barrier synchronization with the test of whether routing on the data network has completed, so that no processor abandons its effort to receive messages until all processors are done sending them.

The control network also detects certain kinds of communication errors and distributes them throughout the system. For example, if two processors attempt to perform different combining operations, an error is signaled. More importantly, hard errors detected by the data network and the network interfaces are collected by the control network. These error signals are combined using a

[4] In dataflow machines, barriers must be programmed in software. A. Shaw *et al.* found that on the EM-4 dataflow machine, barriers could be effectively programmed in software [SKS*92]. Software barriers suffer from unpredictable performance, however. Depending on the other traffic in the data network, software barriers take different amounts of time. It might be difficult to use software barriers for the kind of flow control we study in Chapter 3.

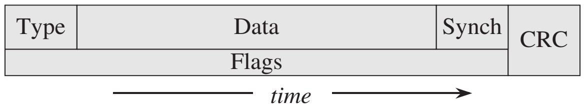


Figure 2-5: The format of messages in the control network. Each message contains a field that indicates the type of message, a 32-bit word of data, some synchronization bits, and various other flags. The message is checked using a cyclic redundancy code.

logical OR and are redistributed to all the processors so that the operating system can isolate them and recover if possible.

Organization of the control network

The architecture of the control network is that of a complete binary tree with processing nodes, control processors, and I/O channels at the leaves. When a CM-5 system is configured, each user partition is assigned to a subtree of the network. Processing nodes are located at the leaves of the subtree, and a control processor is mapped into the partition as an additional leaf.

The control network is implemented using a 1-micron CMOS standard-cell chip that contains custom macro cells to implement high-performance circuitry. Like the data network chip, it uses a 40-megahertz clock. Three binary-tree nodes are packaged on each chip. There are 4 11-bit-wide bidirectional links (6 bits in the up direction and 5 bits in the down direction) to 4 child chips lower in the tree and 1 11-bit-wide bidirectional link to a parent. As in the data network, interchip signals are sent on differential pairs of wires.

Unlike data network packets, control network packets have a fixed length of 65 bits. (There is actually, in addition, a 5-bit packet used during system initialization to align the 65-bit packet boundaries so that a node can process the same fields in arriving messages at the same time.) The general format is illustrated in Figure 2-5. It is broken into two parallel streams, a major stream and a minor stream. The minor stream contains a variety of control bits, including various error and status flags, several flow-control bits, and a bit to implement segmented scans. The major stream begins with a packet description field, which defines the packet type—*single-source*, *multiple-source*, *idle*, or *abstain*—as well as the specific operation—user broadcast, supervisor broadcast, interrupt, scan (including combiner), reduce, etc. Then comes a 32-bit word of data. The major stream ends with a field containing the global synchronization bits. The entire packet is checked using a cyclic redundancy code (CRC), which is the last information in the packet to be transmitted.

The four packet types are processed differently by the control network. Whereas single-source packets are used to implement broadcasting, scans and reductions employ multiple-source packets. Idle packets are used as "filler" and are sent when a control network node has nothing better to ship. The abstain packet allows a control network node to proceed when it would otherwise wait for a multiple-source packet.

When a processor initiates a broadcast or interrupt through the control network, its network interface inserts a single-source message into the tree at a leaf. This message proceeds up to the root node of the user's tree, where it is turned around and distributed to all the processors in the partition. An error is signaled if two single-source packets from different sources meet at a control network node. If it meets with other kinds of packets, a single-source packet has priority. There is no buffering for single-source packets. Flow control for single-source packets is implemented by the network interface on an end-to-end basis.

Processing multiple-source packets is more involved. When a processor initiates a cooperative operation such as a scan, the network interface inserts a multiple-source message into the tree. At each internal node, a multiple-source message waits until its sibling's message has arrived. While a message is waiting, the node sends idle messages up the tree. When the sibling's message arrives, arithmetic or logical operations combine the two messages into one, which is sent up the tree. To implement scans, the message or its sibling may be put aside in another buffer to combine later with a value coming from the node's parent. When a multiple-source message finally reaches the root, it is sent downward. As it reencounters the internal nodes of the tree, it is replicated or further combined with waiting messages. (A good overview of the implementation of scans can be found in [Ble90].)

While a multiple-source packet is waiting for a sibling or a parent, other packets arriving on the same input can be processed. If the newly arriving packet is a single-source packet, it proceeds ahead of the waiting packet, thereby giving priority, for example, to supervisor broadcasts and interrupts. If the new packet is another multiple-source packet, it is queued in the buffer behind the packets already waiting. Multiple-source packets thus maintain a consistent order, which allows two or more combining operations on the control network to be pipelined properly. Flow control in the network precludes buffers from overflowing.

An important requirement of the control network was that it be able to connect a control processor to each user partition. The control processor executes the scalar part of the data-parallel code, while the processing nodes execute the parallel part. We considered having scalar code executed by one or all of the processing nodes, but eventually decided that having a control processor associated with each partition would simplify matters. First, since the system cost of the control processor is very low compared with the multitude of processing nodes, we can afford to run it with large amounts of memory and with additional architectural features to enhance its performance. Consequently, the control processor is able to more efficiently execute scalar code than can a processing node. Second, the data-parallel code that runs on the earlier CM-2 machine is already split into scalar and parallel parts. Porting this code to the CM-5 was easier, since we could maintain the same split. Finally, since the control processor has a connection to an Ethernet, the user partition can run a standard Unix which communicates across the attached Ethernet.

At the end of a user's timeslice during timesharing, the control network can be flushed in a manner similar to a broadcast operation, aborting any user-level control-network operations in progress. The network interfaces retain the values that the user has pushed into the control network until the corresponding operation has completed, however. These values are saved as part of the user's state. When the user's task is resumed, the saved values can be used to reinitiate the control-network operations.

In case of a fault in a CM-5 processing node, network chip, or interconnection link, the control network—like the data network—can be configured to map the fault out of the system. The diagnostic network (see Section 2.4) can set internal switches within the control network to map out parts of the control network. Since the computations performed by the control network depend only on the control network being a binary tree, and not on its being a *complete* binary tree, computations within the control network can safely ignore the mapped-out portions of the system.

In addition, the control network has some additional switching capability to map around faults in the control network itself and to be able to connect any of the control processors to any partition. This additional switching capability is implemented as follows. Conceptually, each switch of the control network has 2 parents and 4 children and contains two binary-tree nodes which can be statically configured so that either can connect to any pair of children. By connecting these chips in a manner similar to the data network fat-tree, any control processor can be connected to any partition, subject to the availability of bandwidth. For example, if there are only 4 control network channels into a subtree, one cannot connect 5 control processors to 5 partitions in the subtree. Short of this bandwidth restriction, however, any connection of control processors to legal partitions can be implemented using an off-line routing algorithm similar to that in [Lei85, Theorem 1].

As an extension to the control-network functionality, one may wish to allow the synchronization of arbitrary subsets of processors rather than the entire set of processors. A control network that employs Ranade's algorithm [RBJ86] could support such an operation. According to standard VLSI complexity arguments (see, for example Thompson [Tho83]), a network that can synchronize, in parallel, arbitrary subsets of processors must either run slowly or have a large amount of wiring.

It is possible to design an inexpensive network that can quickly synchronize any contiguous set of processors in the linear ordering of the processors, however. Such a feature could be used to extend the data-parallel programming model, by allowing one to create an array of data-parallel engines, each of which works on a subproblem, and then to reform the array into one big data-parallel engine to proceed with the top-level problem.[5] There is no clear evidence that hardware support for such a programming model is important, and the CM-5 control network does not support the synchronization of subsets subsequences of processors.

In the CM-5, each partition has a distinguished *front-end* processor. Conceivably, the partition could do without the front-end processor. The distinguishing characteristic of the front-end processor is that it has more memory, a better serial I/O system, and a faster processor than the rest of the processors. It is often convenient to have this more-powerful processor take on a special role in controlling the computation by, for example, using the control network to broadcast the state of the computation or using the large memory of the front-end to hold large infrequently accessed data. A front-end on the CM-5 is especially useful for running software that was written for other machines that had front-ends. For example, the StarTech chess program (described in Chapter 4), which is based on H. Berliner's Hitech program, uses the CM-5 front-end to run the code that runs on Hitech's front-end processor.

In summary, the CM-5 control network provides the mechanisms to allow data-parallel code to be executed efficiently, as well as allowing more general kinds of parallel models to be implemented. Its structure as a binary tree provides an inexpensive way to provide the advantages of both traditional SIMD and traditional MIMD architectures.

2.4 The CM-5 Diagnostic Network

During the design of the CM-5, great emphasis was placed on system availability. Despite conservative design techniques and the use of proven circuit and interconnect technologies, the sheer size of the largest CM-5 systems forced us to abandon any attempt to achieve high availability by depending solely on inherent component reliability. Instead, our strategy relies on two architectural features of the machine: diagnosability which allows missing or broken hardware to be detected and isolated; and configurability, which allows most of the machine to operate when portions are broken or being serviced. This section shows how this strategy is implemented on the CM-5 through the use of a diagnostic network, the one network in the system that the user never sees.

One strategy to diagnose a parallel computer is to create diagnostic programs running on the processor nodes that exercise the processor nodes and various communications networks. When some part of the system fails to function correctly—for example, the data router fails to deliver a message or the control network produces the wrong answer for a combine operation, the diagnostic

[5] Sabot's paralation programming model [Sab88] may provide a way to program virtual data-parallel machines.

program itself may fail, because its correctness depends on the correct functioning of the system. We call such diagnostic programs *functionality dependent*. Our experience with the CM-1 and CM-2 exposed many of the limitations of functionality-dependent diagnostics. They are exceedingly difficult to write, they have nebulous coverage, and they lack precision in reporting the root cause of error conditions.

In contrast, diagnostics that are *functionality independent* rely on specific test structures, rather than the failure of normal system operation, to detect faults in the system. Using this kind of design-for-testability strategy, it becomes possible to view the CM-5 (or *any* sequential machine, for that matter) in terms of registers connected by combinational logic and wires. This change in perspective permits commercially available software tools to be used to generate high-coverage tests automatically for chips, boards, and the wiring that connects them. Moreover, when these tests fail, they provide specific information on the location and extent of the failure.

In the CM-5, design for testability starts at the chip level. All CM-5 VLSI components support the IEEE 1149.1 testability architecture standard [IEEE90], also known as JTAG, for the Joint Test Action Group which originated the standard.[6] At the system level, the CM-5 diagnostic network provides parallel access to all system components from a *diagnostic processor*. The JTAG standard and the diagnostic network combine to form a diagnostic system which can quickly perform an in-system check of the integrity (over 99 percent single stuck-at fault coverage) of all CM-5 chips that support the JTAG standard and all networks.

Let us briefly review the JTAG interface standard. The JTAG standard provides a 4-pin interface for each chip in a system. On each chip, two pins provide input and output, respectively, for a selectable scan chain within the chip.[7] The standard specifies the *boundary scan register* (BSR) which connects all I/O pads in the chip into a bit-serial shift register. Two other pins serve as clock and control inputs. By scanning data in and out of chips, the BSR can be used to apply stimulus to the chip core for chip tests, or to monitor inputs and control outputs of the chip for connectivity tests.

In the CM-5, we extended the JTAG standard to include full internal scan in all proprietary chips. Details of this design are described in [ZH92]. The use of a full internal scan allows software for automatically generating test patterns to generate a set of scan vectors with very high fault coverage. The vectors can be applied through the JTAG interface to test individual chips when they are manufactured and packaged. Later, when the chips are assembled into a system, the same tests can be applied through the diagnostic network.

The JTAG interface is designed to extend to multichip systems. When more than one chip is incorporated in a system, the scan paths are linked together in series by connecting the output from one scan path to the input of the next in a daisy-chain fashion. The clock and control pins are connected in parallel so that these signals can be broadcast to all chips in the chain.

Previous designs have focused on reducing the length of very long scan chains by placing scan-controllable bypass elements in the scan chain [TI90]. Unfortunately, testing all the chips in the system still requires serial access to each one. Even with ideally short test times on the order of seconds per device, this method would be unacceptably slow for an entire 16,384-node CM-5 comprising many tens of thousands of devices. Moreover, this method fails to take advantage of the

[6] In the original implementation of the CM-5 architecture, neither the SPARC processor nodes nor the DRAM chips supported the JTAG interface. Given the growing acceptance of JTAG standard, however, it is likely that off-the-shelf processors and memory will support the standard in the near future. The CM-5 architecture is designed to incorporate these JTAG-supporting chips when they become available. For example in the CM-5E, announced in February 1994, the SuperSPARC processors do support the JTAG interface.

[7] This use of the term "scan" has nothing whatsoever to do with parallel prefix and suffix computations, as discussed in Section 2.3.

inherent parallelism that can be achieved by testing large numbers of identical system components. For these reasons, it was evident early on in the design of the CM-5 that we needed a parallel strategy for supporting scan-based diagnostics.

The CM-5 diagnostic network provides simple and reliable access to the system components of the CM-5. It provides scan access to all chips supporting the JTAG standard, and programmable *ad hoc* access to non-JTAG chips. The diagnostic network itself is completely testable and diagnosable. The diagnostic network is able to map out and ignore parts of the machine that are faulty or powered down. It can be partitioned consistently with user partitions. The network is able to select and access groups of system chips in parallel, including:

- a single chip;
- a single type of chip;
- the chips within a user partition;
- the chips associated with a geographical portion of the system, *e.g.*, a given board, backplane, cabinet, etc.; and
- unions and intersections of previously specified sets of chips.

The diagnostic network is organized as a (not necessarily complete) binary tree, at the root of which sit one or more diagnostic processors, and at the leaves of which are *pods*. Each pod is a physical subsystem, such as a board, which directly supports the JTAG interface. At any given time, a single diagnostic processor controls the diagnostic network. From the root of the tree, an individual pod can be addressed by giving a binary number, each bit of which corresponds to a level in the tree and specifies a path from the root to the leaf: bit i of the address specifies whether the addressed leaf is in the left or right subtree of the node at level i. If the height of the tree is h, then h bits are sufficient to specify any leaf.

The diagnostic network allows groups of pods to be addressed according to a "hypercube address" scheme. For a tree of height h, a *diagnostic virtual address* is an h-digit number in which each digit is a `0`, `1` or `B`. The `B` ("both") digit is a "wild-card" that matches both `0` and `1`. For example, in a height-6 tree, the address `00B10B` addresses the set $\{\texttt{000100}, \texttt{000101}, \texttt{001100}, \texttt{001101}\}$, or $\{4, 5, 12, 13\}$. The addressing scheme can also be used to address the internal nodes of the diagnostic network by specifying addresses with fewer than h digits.

The decoding logic to implement the diagnostic virtual addressing scheme is based on the notion of steering "tokens" down the tree, as is illustrated in Figure 2-6. The mechanism works as follows. A token is inserted at the root of the tree together with a diagnostic virtual address, which is piped digit-serially into the root of the tree, high-order digit first. The root selects its right, its left, or both of its subtrees based on the high-order digit. If both subtrees are selected, the token splits into two tokens. Subsequent digits then steer the tokens and subsequent digits down the selected paths. When the end of the address is encountered, the nodes holding tokens are considered to be selected, and nodes on paths from them to the root provide the conduit for control.

Tokens and their paths from the root stay in place until a subsequent address erases them or until they are explicitly erased. This feature can be employed to combine two sets of selected nodes. For conceptual simplicity, suppose each of the two sets of nodes is in a separate subtree of the root. First, the left set is selected using a 0 as the high-order digit and pushing a token down the appropriate paths. Next, the right set is selected using a `1` as the high-order digit and pushing a token down the appropriate paths. The left set remains intact, but is temporarily inaccessible from the root because the right set is being selected. Finally, we push another token with an address of `B` to select the root itself and cause it to enable both its children, thereby merging the two sets. More complicated set unions are possible using this basic mechanism.

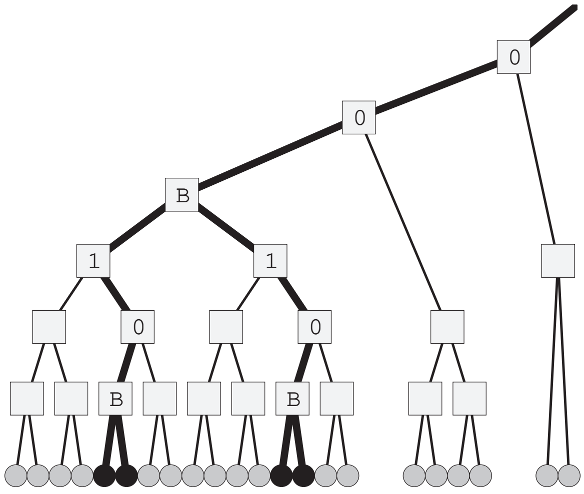


Figure 2-6: Steering a token down the diagnostic network. The address is decoded digit-serially, where each digit is `0`, `1`, or `B`, representing a selection of the left subtree, right subtree, or both subtrees, respectively. The example shows the selection made by the address `00B10B`.

Most of this mechanism is hidden from the diagnostic engineer. Software extends the diagnostic virtual address within pods to address individual chips. Software also converts between the diagnostic network addresses and two other kinds of addresses: *geographical addresses*, which specify cabinets, backplane, slot type, slots, etc.; and *network addresses*, which give the locations of components according to the data and control networks' view of the machine. In general, important subsets of geographical addresses can be specified with one diagnostic virtual address. Important subsets of network addresses—for example, all data network chips at a given height in the machine, or all boards containing processing nodes in some contiguous range—typically take a combination of at most h diagnostic virtual addresses, where h is the number of bits in the address. The most important aspect of the addressing scheme, however, is that the time to access the various subsets does not grow by more than a small additive amount when the size of the machine doubles.

Having addressed a subset of the pods in the system, scan vectors can be applied in parallel to detect errors. JTAG serial data and control inputs are broadcast to all selected pods. Each pod provides a scan output signal that can be OR'ed or AND'ed with the corresponding signals from the other selected pods. The choice of an OR or AND combiner depends on what the diagnostic processor is expecting for a scan result. If the expected bit is a 1, the AND combiner is chosen. The result of the combining is a 1 if and only if all selected pods assert a 1. Similarly, if the expected output is a 0, the OR combiner is chosen. The result of the combining is a 0 if and only if all selected pods assert a 0. If an error is detected in a group of selected pods, the offending pod can be isolated either by addressing each pod in the group individually one at a time, or by a divide-and-conquer methodology. Within a pod, standard techniques for finding errors within a serial chain of JTAG interfaces are used to isolate the error to the chip level.

Since the diagnostic network is a tree, it is relatively easy to make it self diagnosing. Each level beneath the root can be tested by the levels above. Moreover, since there is not much logic in the diagnostic network, the probability that the network fails itself is much less than the probability that other parts of the system fail. Moreover, since the network is a tree, most of its logic is near the leaves, so that when a part of the diagnostic network does fail, only a small part of the tree is likely to be isolated. We did not mind relying on relatively few components near the root, since any small set of components is quite reliable—it is only large aggregates which have a high probability of failing.

The current implementation of the diagnostic network uses essentially two off-the-shelf chips. The address decoding of a binary node is implemented with a P22V10 24-pin PAL, and the finite-state control of a node is implemented with a P18V8 20-pin PAL. The chips can be clocked at any speed up to about 1 megahertz. In some places in the system, to save chips, address decoding of a 4-ary or 8-ary node is implemented directly as a single-chip PAL, rather than by using several separate binary-node PAL's.

2.5 Synchronized MIMD Goals

When we first set about designing the CM-5, we established engineering goals that went beyond mere performance specifications. We thought hard about issues of *scalability*: making a machine whose size would be limited only by the dollars a customer could spend, not by any architectural or engineering constraint. We thought hard about system issues, including timesharing, I/O, and user protection. We thought hard about reliability, since we were designing a machine which, in its largest configuration, would have well over 10 times the electronics of our previous supercomputer, the Connection Machine Model CM-2 Supercomputer. This section discusses the goals and reflects on the success of the CM-5 at meeting those goals.

The following goals drove our network designs:

- The networks must deliver high performance to the users. We wanted the users to be easily able to program the networks to get good performance. We did not want to force the users to worry constantly about pathological worst cases, and we wanted the best cases to run well without the user needing to do anything special.
- The networks must scale up to a very large size. We wanted the logical design of the networks to scale up to a million processing nodes. We wanted to build SUPERcomputers.
- The networks should efficiently support the data-parallel programming model (see Section 2.3), but should be flexible enough to allow us to support other parallel programming models as well. The data-parallel programming model was used extensively on the CM-2, and we wanted to be able to transport our existing high-level programming environments (Fortran, *Lisp, and C*) to the CM-5. We also wanted to be able to run codes written for other machines competitively.
- The networks must be highly reliable and highly available. The system must notice whenever part of a network fails, be able to isolate the failure quickly, and be able to quickly reconfigure the networks around the failure. It was desired that even if part of a network has failed, the rest of the network should be able to function correctly with only a small degradation in performance.
- The networks must work in a spaceshared environment. We wanted a user's network traffic to be insulated from other users and I/O in other partitions.
- The networks must work in a timeshared environment. A timeshared user must get a fair share of network bandwidth. Users must be able to be context-swapped quickly. Privileged system software must be able to seize control of a user's task.
- The networks must be operational as soon as possible. Time to market was of the essence. Chips and systems needed to work the first time. We wanted the networks to be simple enough to engineer quickly, robust enough to respond to last-minute design changes, and easily verifiable. Consequently, we opted for conservative technology, for example, copper wires rather than optical fibers. We chose to use CMOS in order to minimize the risk associated with new technology.

We chose standard-cell technology in order to be able to make extensive use of the wide variety of available design tools (such as timing verifiers and automatic test generators). To achieve high performance with this conservative technology, we incorporated custom macro cells for circuits on the critical path. Our attitude was that there was more performance to be gained by architectural improvements than by eking out extra nanoseconds in technology. Conservative technologies, with their well-developed computer-aided design tools, would allow us to make many more architectural improvements during the design.

- The chips used to build the networks must be organized in a way to allow technological or architectural improvements to be easily incorporated in subsequent revisions of the CM-5 system. On the CM-2, both processors and communication were implemented on the same chip, which made it difficult to incorporate advanced technology in one area without impacting the other. We wanted to be able to incorporate any advances without having to reengineer a major piece of the system.
- The networks should embody both economy of mechanism and single-minded functionality. We wanted the networks to be lean and mean. Whenever anyone suggested anything complicated, we viewed it with suspicion. For example, the job of the data network is to deliver messages, nothing else. But it delivers both user messages and messages to I/O devices using the same mechanisms. The data network does not combine messages, duplicate messages, or acknowledge delivery of messages. It just focuses on moving data as fast as possible.

In hindsight, those goals missed an important part of the story: the operating system. The CM-5 operating system must not only provide the functionality of UNIX, but it must interact with the global scheduling strategy, all-fall-down, and the approach to shared I/O resources. It has been a long and difficult passage to obtain correct functionality and performance. The other side of the coin is that today the CM-5 operating system is the only parallel operating system that even attempts to meet these goals.

The CM-5 not only uses global synchronization for running user code, but it also uses global synchronization to run the operating systems. All of the processes that belong to a single user program running on a single partition are coscheduled so that the network interface does not need to understand about process identifiers. This synchronous coscheduling fitstogether with the all-fall-down mechanism. The all-fall-down mechanism has been difficult for the operating-system programmers to manage, however, chiefly because there is no corresponding 'all-throw-up' mechanism to put the messages back into the router. The operating system needs a strategy to deal with the situation where not all of the messages are successfully reinjected into the data network at the beginning of a process's time slice. Most of the difficulties with programming the all-fall-down mechanism have been dealt-with, but we shall see evidence in Section 3.3 that the all-fall down mechanism is still causing some performance trouble. Perhaps there is a better way of achieving our goals than to start with the assumptions that the network state as part of the process state and that messages are reliably delivered.

The operating system also has difficulty understanding how to use the diagnostic network of the CM-5. Large machines still take too long to boot. The operating system has difficulty accessing the diagnostics network to get the machine configured in parallel. These problems with the operating system are not fundamental to the design of the system, but they are difficult.

The cost of writing large parts of an operating system has been fairly high. The other operating systems that one might use, however, either exist only in the future or do not address the system-wide problems of parallel supercomputing. On most other machines, one does not even think of measuring the system while timesharing is running. For example, at Mannheim, one application on the Kendall Square KSR-1 shared memory sees a factor of 40 times slowdown when timesharing is being used as opposed to single-user [Sch93, Page 44].

When designing the CM-5, we tried to provide some performance guarantees, and the synchronization facilities helped us do that. The performance a user gets from the data network should depend only on the user's traffic in the data network. If the machine is being timeshared, then the behavior of one user should not affect the performance of another. Each user should receive not only a fair fraction of the processing power, but also a corresponding fraction of the data network and control network performance. If the machine is space-shared, so that different users are running on different processors, the user should attain predictable performance independently of what is going on in the other processors. The data network and the control network should not interfere with each other. We specify this architecturally by saying that the synchronization is provided by a separate control network. Operationally, this means that the performance of a global synchronization must be independent of the traffic in the data network. In the CM-5 we actually provide a separate control network, but one can imagine implementations where the logical control network is implemented using the same wires used by the logical data networks.

We might have made some implementation decisions differently if time-to-market had not been quite as important a goal. For example, one should be able to achieve better performance by implementing the two logical data networks as a single physical network with two virtual subnetworks. Similarly, the control network is not as general as it might have been. The control network can not be segmented to allow, for example, the user to create in a single application a collection of small synchronized-MIMD machines that operate in parallel, as was proposed for the FMP [LB80].

2.6 CM-5 History

We conclude this chapter with a brief history of our implementation effort.

Work on the CM-5 architecture was begun in the latter part of 1987. We performed network simulations that led us, by January 1988, to choose a fat-tree architecture for the data network. By May 1988, most of the data network logic had been designed and verified, although several changes were implemented during the summer of 1988. A register-transfer-level (RTL) description of the data network chip was completed in early 1989, and the data network architecture was frozen. A gate-level description of the data network chip was completed by the early summer of 1989. The JTAG diagnostic interface was debugged using the data network chip design as a framework. The data network chip also served as the guinea pig for system and chip timing software. The chip was submitted for fabrication in May 1990.

The MIMD-plus-control-network design was proposed in early 1988, but we did not officially decide to use it until May 1989. Until then, we maintained other potential design alternatives. Work on the control network chip and the network interface proceeded concurrently. By the end of summer 1989, RTL models of both were simulating successfully. Gate-level models were implemented by the end of December 1989, and the control network architecture and network interface were frozen shortly thereafter. In May 1990, both the control network chip and the interface chip were submitted for fabrication.

The strategy of the diagnostic network was laid out in 1988, but work did not begin on it in earnest until the fall of 1989. Most of the work involved implementing the JTAG interface on the various chips. The design of the diagnostic network itself took only a few months, but considerable effort in 1990 and 1991 went into diagnostic software.

In the latter part of 1990, our attention turned to system integration. We received and tested the data network chips in July 1990, the control network chips in August, and the interface chips in September. Within two days after the interface chips arrived, we had assembled the networks for a 2-node machine and powered it up, a feat due in large measure to our functional verification methodology [SYC92]. That same day, the operating system—which had been developed on a simulator—functioned correctly on the machine. By year's end, we had successfully constructed several small machines, including a 64-node machine, some of which were dedicated to software development.

The year 1991 began with an effort to build a 256-node machine using a completely new mechanical design. Initially, it had been more important to make machines available to our software engineers than to construct a large machine. To test the limits of our physical design, however, we needed to build large machines. The 256-node machine was begun in February, and finished in March. The time frame was dominated by the build time in manufacturing. In May, we built a 544-node machine, which was shipped in August to the Minnesota Supercomputer Center on behalf of the Army High Performance Computer Research Center.

In October 1991, the Connection Machine Model CM-5 Supercomputer was publicly announced. During 1992 a 1024-node CM-5 was constructed and installed at Los Alamos National Laboratory, and the machine passed acceptance testing during February 1993. March 1994 saw the announcement of the Connection Machine Model CM-5E, which provides higher data network performance using larger, 68-byte messages; an improved processor-network interface; and faster processing nodes that employ SuperSPARC microprocessors and faster vector units, both running on a 40 megahertz clock.

Since supercomputer manufacturers tend to keep their sales figures secret, it is difficult to gauge the commercial success of the CM-5. J. Dongarra, H. Meuer and E. Strohmaier [DMS93] estimate that Connection Machine computers (including both the CM-2 and the CM-5) collectively account for about 30% of the peak LINPACK capacity of and about 40% of the peak floating-point capacity of the fastest 500 supercomputers installed in the world in early 1993. G. Ahrendt's list of the world's most powerful computing sites [Ahr94] states that there probably exist at least the following CM-5's: one with 1056 processing nodes, one with 896 nodes, three with 512 nodes, one with 256 nodes, one with 192 nodes, and six with 128 nodes.

Chapter 3

Mechanisms for Data Network Performance[1]

3.1 Introduction

We have now studied hardware (the Connection Machine CM-5) that provides global synchronization, and have seen how global synchronization solves many of the system-problems in the CM-5. In this chapter we study how to get good performance from the data network of the CM-5. Once again, global synchronization provides a simple and effective strategies for attacking the problem.

Suppose you need to perform a sequence of parallel cyclic shifts. In parallel FORTRAN you might see code that looks like this, where the columns of `A` and elements of `B` are distributed among the `P` processors:

```
DIMENSION A(P,P), B(P)
A(1,:)  = CSHIFT(B,1)
A(2,:)  = CSHIFT(B,2)
...
A(P,:)  = CSHIFT(B,P)
```

The notation "`A[I,:]  = CSHIFT(B,I)`" means cyclic-shift `B` by `I` and store it into row `I` of `A`.

One natural way to compile data-parallel code is to compile each statement separately with global barriers between the statements. In this case, both the source language and the compiled code are a serial sequence of parallel operations. A simple subscript analysis reveals that the barriers are not required to ensure the semantic correctness of this program, since the target rows are all independent.

Question: What happens to the performance when the barriers are removed?

Answer: Surprisingly, the computation gets slower, often by a factor of three.

To understand this problem, a few patterns were studied in a narrow environment:

- We studied the cyclic-shift pattern described above, as well as resulting all-pairs communication pattern in which every processor sends a value to every other processor. The all-pairs

[1]The research in this chapter represents joint work with Eric A. Brewer. Much of the work of this chapter was originally reported in [BK94].

pattern appears in sorting and in some scientific codes (see, for example [Ede91].) We also studied random communication patterns.

- We studied the communication patterns in a data-parallel or SPMD environment, in which the real operation being performed by a collection of messages is a bulk data movement. Given this assumption, we examined the problem of sending a large collection of messages as quickly as possible, rather than focusing on the performance of any particular message.

- We used block transfers built on top of 20-byte active messages [vCG*92] on the CM-5 data network. (The data CM-5 network is described in Section 2.2 of this dissertation.)

Although the scope of this study has been narrowed from the wider problem of obtaining good performance on any interprocessor communications system, we believe that our conclusions apply to a fairly wide range of situations.

An important limitation in any network is its bisection bandwidth. In general, given a data-parallel communication operation, if you divide the processors of a machine into two sets, and then measure the bandwidth B in bytes per second that the network could possibly provide across the corresponding cut, and you measure the amount of data D in bytes that must be transferred between the sets, then it must take at least D/B seconds to move the data. For any given cut, B is a function only of the network, and D is a function only of the communication pattern.

There are 2^{P-1} ways to cut P processors into two sets, which makes finding the worst cut a potentially formidable operation.Leiserson [Lei85] showed that for fat-trees the problem is much easier: one need only consider the cuts across a single major arm of the fat-tree in order to find the tightest D/B bound (see Figure 3-1). In a P-node CM-5, which is a uniform 4-ary fat-tree, there are less than $4/3P$ major arms, and the bandwidth of an arm depends only on the height of the arm. The bandwidths for a 64-node CM-5 are shown in Figure 3-1. We found that the most important cuts for the CM-5 are the links that connect the processors to the network: the bandwidth of these links are determined by software overhead and form the limiting factor for most message patterns.

We found that programmers of the Connection Machine CM-5 data network can improve the performance of their data movement code more than a factor of three by using a few relatively simple mechanisms.

- Selectively using global barriers eliminates pathological interactions between different parts of the computation and provides a form of flow control. Barriers improve the performance of cyclic shifts by a factor of 2 to 3.

- Managing the order in which messages are injected into the network improves the statistical independence of the various packets that are in the network at any given time, which avoids worst-case performance scenarios. If a processor has many packets to send to each of several processors, it is better to interleave the packets to several destinations rather than send large batches of packets to one target. For cyclic shifts, such a strategy is worth a factor of 2.1 (but that factor is not independent of the factor gained by using barriers.) More generally, injecting the messages into the network in a random order is a good idea.

- Limiting the rate at which messages are injected into the network provides another form of flow control which helps to reduce network overloading and improve performance. For cyclic shifts, this optimization is worth an additional 25% in performance, and it greatly reduces the variance in bandwidth for large transfers.

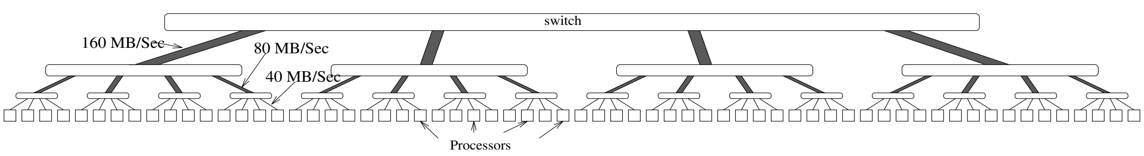


Figure 3-1: A 64-node CM-5 data-network fat-tree showing all of the major arms and their bandwidths (in each direction). One needs to cut only a single major arm to find the worst bisection for a given message pattern.

This chapter explores how to get good performance from the CM-5 data network. Section 3.2 reviews the CM-5 networks with an eye to performance. Section 3.3 discusses the difficulties of measuring performance on the CM-5. Section 3.4 examines how barriers can improve communication performance, and Section 3.5 explores the importance of interleaving packets to multiple destinations. Section 3.6 examines the effect of matching the injection and reception rates. Section 3.7 concludes by outlines some programming rules of thumb for using these mechanisms.

3.2 CM-5 Background

This section provides some background on the CM-5 data network and examines the fundamental limitations of the machine, including network-processor bandwidth and network capacity.

Recall from Chapter 2 that the CM-5 data network is a 4-ary fat-tree, as shown in Figure 3-1. Each edge is actually two independent links, left and right, but for bulk data movement we always use both simultaneously. Of the various network cuts, at least two matter in practice: the links connecting the processors and the cuts through the root.

Processor overhead limits the bandwidth of the processor-network links, not the network hardware. For these links, the hardware can support up to 40 megabytes per second in each direction. Assuming the 33-megahertz clock found in most CM-5 implementations and 20-byte packets with 16 bytes of payload (also standard for the CM-5), the sending overhead out of the cache is at least 37 cycles, for a maximum *payload bandwidth* of $(16 \times 10^{-6})(33 \times 10^{6})/37 = 14.3$ megabytes per second.

The real limit is the cost of receiving packets, which currently requires about 60 cycles for realistic packets with polling and requires hundreds of cycles with interrupts. Because of the prohibitive cost of interrupts, all of our experiments use polling. At 60 cycles per 16-byte packet, the payload bandwidth is limited to 8.8 megabytes per second. Kwan, Totty, and Reed [KTR93] measured the actual one-way bandwidth at 8.3 megabytes per second using Thinking Machines' message-passing library.

These numbers only cover the case in which a processor is sending or receiving, however. When a processor is both sending and receiving, the bidirectional bandwidth is somewhere between the two cases. Although the network handles both directions in parallel, the processor cannot. The overhead to send and receive a packet is about 90 cycles, saving 7 cycles due to shared code. This translates to an upper bound of 5.9 megabytes per second in each direction for a total of 11.8 megabytes per second. The largest value measured by Kwan *et al.* was 10.4 megabytes per second.

The *capacity* of the network, which is the number of packets that can be injected without the receiver removing any, limits the ability of the processors to work independently. For example, if the network can hold ten packets, then a processor can inject only ten packets before the network backs up, and then it must wait for the receiver to accept packets. The network capacity may be a function of the message pattern. For the CM-5, we measured the network capacity for a variety of partition sizes using cyclic shift by half the machine size:

Nodes	Total Packets	Packets/Node
8	79.0	9.88
16	158	9.89
32	342	10.7
64	691	10.8
128	1441	11.3

Thus, for any substantial data movement, the senders and receivers must be coordinated. Furthermore, since only the wire time can be hidden and the network capacity is only about eleven packets, there is little profit in trying to overlap computation and communication on the CM-5.

Nearly all communication on the CM-5 is implemented with active messages. Active messages were developed by von Eicken *et al.* [vCG*92], whose Berkeley CMAM package provided substantially better performance than contemporary versions of CMMD, Thinking Machines Corporation's communication library. CMMD 3.0 incorporated the active-message ideas, and in fact, most of CMMD is now implemented via active messages. Like CMAM, CMMD provides support for barriers and block transfers.[2]

Before active messages were developed, most MIMD parallel computer systems provided active libraries based on the the crystalline message passing model described by G. Fox [Fox89]. In such a model, for two processors to communicate, one must perform a *send* and one must perform a *receive*. In the synchronous form, the send and the receive are blocking — the send blocks until the corresponding receive is executed and only then is data transferred. Since data is only transferred after both its source and destination are known, no extra buffering is needed at either the source or the destination processors. To implement the matching of the send and the receive requires a three phase protocol, in which the sender sends a packet requesting permission to send, the receiver replies with a packet containing the permission, and then the sender finally starts sending the data. To hide this latency, most systems provided a *non-blocking send* operation, but such schemes usually require quite a bit of buffering. For a more detailed comparison of active messages to other message passing paradigms, see [vCG*92]. Even the early versions of CMMD on the CM-5 were based on the crystalline model.

As part of the CM-5 design team, I helped implement all kinds of hardware mechanisms that can be used to obtain good performance. What we designers sometimes did not adequately appreciate was that each of those mechanisms had to be programmed by some human. Furthermore, even though we designers tried to think of everything that would be needed, some issues that we never imagined have turned out to be important, such as the difficulty of implementing a parallel operating system (see the discussion of this in Section 2.5). The fact that CMMD still does not provide split phase barriers (as of Spring 1994) is an indication that these kinds programming problems are still present.

The Strata communications library [BB94b], developed at MIT, is an alternative to CMAM and CMMD that provides improved performance, improved support for timing and debugging, precise control over polling, and split-phase control-network operations.[3] Strata incorporates the techniques described in this paper.

[2]The CMMD performance numbers presented in this chapter were measured under CMMD 3.1-Final with CMOST 7.2-Final.

[3]Strata is available from `ftp.lcs.mit.edu` via anonymous ftp, directory `/pub/supertech/strata`.

Network Status	Average	95% CI
Empty	4.56 seconds	± 0.0020
Full	5.52 seconds	± 0.24

Figure 3-2: The effect of a full network on timings made with the operating-system timers: the timers inflate timings 21% when the network is full.

3.3 Timing on the CM-5

We use two forms of timing depending on the expected length of the event. For short events, less than a millisecond or so, we use the 32-bit cycle counter. For longer events, we use the 64-bit timers provided by the operating system.

The cycle counter is extremely accurate. By using inline procedures the overhead can be subtracted out to yield timings that are accurate to the cycle. The cycle counter counts everything including interrupts and other processes, however. For example, if our process is time-sliced during an event, then we count all of the cycles that elapse until we are switched back in and the event completes. The probability of getting switched out during a one-millisecond event is about 1 in 100, however, since the time-slice interval is one-tenth of a second. A more common problem is the timer interrupts, which occur every 60th of a second.[4] Thus, to get reliable measurements, we usually perform a timing at least three consecutive times and take the median. Using the median effectively eliminates errors due to time slicing and timer interrupts.

The operating-system timers have their own advantages and disadvantages. The operating-system timers stop running when your process stops running, so they can be used even across time-slice interrupts. The operating-system timers are accessed using system calls, however, which cost hundreds of cycles and thus limit the accuracy. The operating-system timers also experience a time dilation when the data network is full of messages.

To demonstrate the time dilation of the operating-system timers, we ran the following experiment. We timed a floating-point loop with the network empty, filled up the network, and timed the same loop with the network full. The only difference between the "empty" time and the "full" time is the presence of undelivered messages sitting in the network. The floating-point code is identical, no messages are sent or received during the timing, there are no loads or stores, and the code is a tight loop to minimize cache effects. Figure 3-2 shows the results for 18 samples taken across a wide range of overall system loads. Not only does filling up the network increase the measured time to execute the floating-point loop by an average of 21%, but it substantially increases the variation in measured time as well, as shown by the wider 95% confidence intervals.

This study implies that timings that occur while the network is full are dilated an average of 21%. The dilation is load dependent, but we were unable to get a reliable correlation between the dilation and the average system load. Fortunately, the timings appear to be consistent given a particular mix of CM-5 jobs, and the inflation appears to change slowly with time. To obtain reliable data, we ran the set of all experiments twelve times, measuring all of the experimental configurations once, and then measuring them all again, and so on, so that slow changes to the environment tend to affect all of the experiments equally. Algorithms that keep the network full appear to achieve lower performance than algorithms that keep the network empty. We address the time-dilation issue in the context of each experiment.

[4] The timer interrupts take about 250 microseconds to complete, which means that a CM-5 (of any size) spends about $250/16666 = 1.5\%$ of its cycles handling timer interrupts.

We believe that the dilation is caused by context-switching. At each time slice, the operating system empties the network, using the all-fall-down mechanism, so that messages in the network that belong to one process do not affect the next process. When the process is switched back in, its messages are reinjected before the process continues. The cost of context switching appears to depend on the number of packets in the network, and some or all of this cost is charged to the user and thus affects the timings. Our measurements have not been adjusted for time dilation, since it appears that algorithms that keep the network full really do run slower. Our experiments with bulk data movement indicate that algorithms that keep the network full suffer more from network congestion than from time dilation.[5]

Even though the CM-5 operating system suffers from time dilation of 21%, we still consider the situation to be a victory for the cause of predictable performance. The fact that we can measure a 21% slowdown due to timesharing is in fact a victory. Some machines, such as the KSR1 see a factor of 40 slowdown when using timesharing [Sch93, Page 44]. The CM-5 provides relatively predictable performance under timesharing and is quite close to the goal of totally predictable performance under timesharing.

3.4 Using Barriers Can Improve Performance

This section shows that adding barriers to a communications operation can actually increase performance, and presents some evidence to explain the benefit. To our knowledge this effect was first noticed by Steve Heller of Thinking Machines Corporation [Hel92]. Culler *et al.* mention the effect in a later paper [CKP*93].

We ran the cyclic-shift experiment as follows. On a 64-node CM-5, each processor sends a total of 1.28 megabytes using block-transfer primitives. Each processor p sends a block of data to processor $(p + 1)$ mod 64, then sends a block to $(p + 2)$ mod 64, and so forth. We vary the block size B. For example, when $B = 0.02$ megabytes, each processor sends exactly one block to each processor; and when $B = 0.01$ megabytes, each processor cycles around twice, on each *round* sending one block to each processor. Figure 3-3 shows the performance of this cyclic-shift pattern as we vary B for both Strata and CMMD 3.1, with and without barriers.

The versions with barriers use a barrier between cyclic shifts; i.e., each processor sends B bytes, waits for the barrier, and switches to the next destination. The CMMD version with barriers must use Strata's barrier procedure. Unlike Strata's barrier, the CMMD barrier does not poll, and hence combining it with block transfer leads to deadlock. You could use CMMD's barrier if the system used interrupts instead of polling, but the loss due to interrupt overhead is prohibitively expensive.

Except for very small blocks, the versions with barriers perform much better. At 64-byte blocks (only 4 packets) the difference is small, but by 128 bytes per block, the difference is roughly a factor of two. For larger blocks, which are the common case, the difference is about a factor of 2.5. The substantial drop in bandwidth without the barriers is counterintuitive. Removing the barriers reduces the overhead and provides the data network with more opportunities to route packets. The increased opportunity, however, translates to decreased performance.

Some sort of interference occurs when packets from different batches interact. We were able to measure an interaction that we call *target collisions*. A target collision occurs when two packets arrive at the same processor at nearly the same time. Since packet reception is the bottleneck, target collisions can quickly back up the network. For large batches, target collisions can conceivably

[5] The use of dedicated mode does not eliminate the dilation, although it does provide a more stable measurement environment.

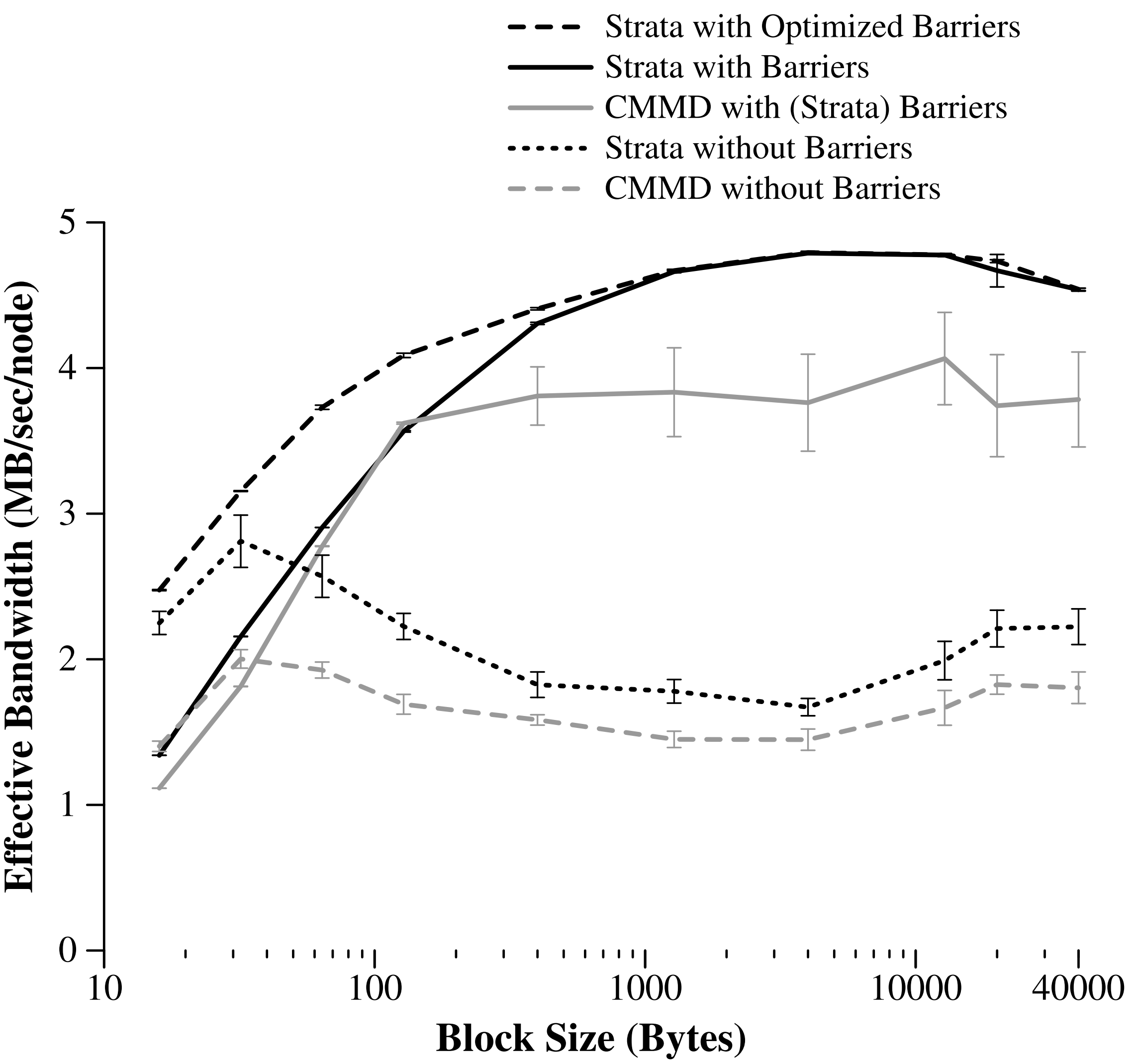


Figure 3-3: The effect of barriers between block transfers on the cyclic-shift pattern. The lines mark the average bandwidth; the error bars indicate 95% confidence intervals.

slow things down quite a bit. For example, if for some reason, two processors each started sending a batch to the same processor at the same time, then the destination processor would be overloaded, the network would back up, and the performance would drop substantially.

To observe target collisions, we measured, at each instant in time, the number of packets in the network that are destined for a given processor. We performed this measurement by recording the time that each packet was injected into the network and the time that the target received the packet. We were able to use the globally synchronous cycle counter to obtain consistent times.

Figure 3-4 shows evidence of target collisions for one typical case: cyclic shifts with no barriers and a block size of 100 packets (1600 bytes). The plot shows for each processor, at each point in time, the number of messages destined for that processor. There are several interesting patterns in this data. At the far left there are some patterns that appear as a “warped checkerboard” pattern around 200,000 cycles. These patterns reflect the propagation of delays. A single packet was delayed, which caused the destination processor to receive packets from two different senders,

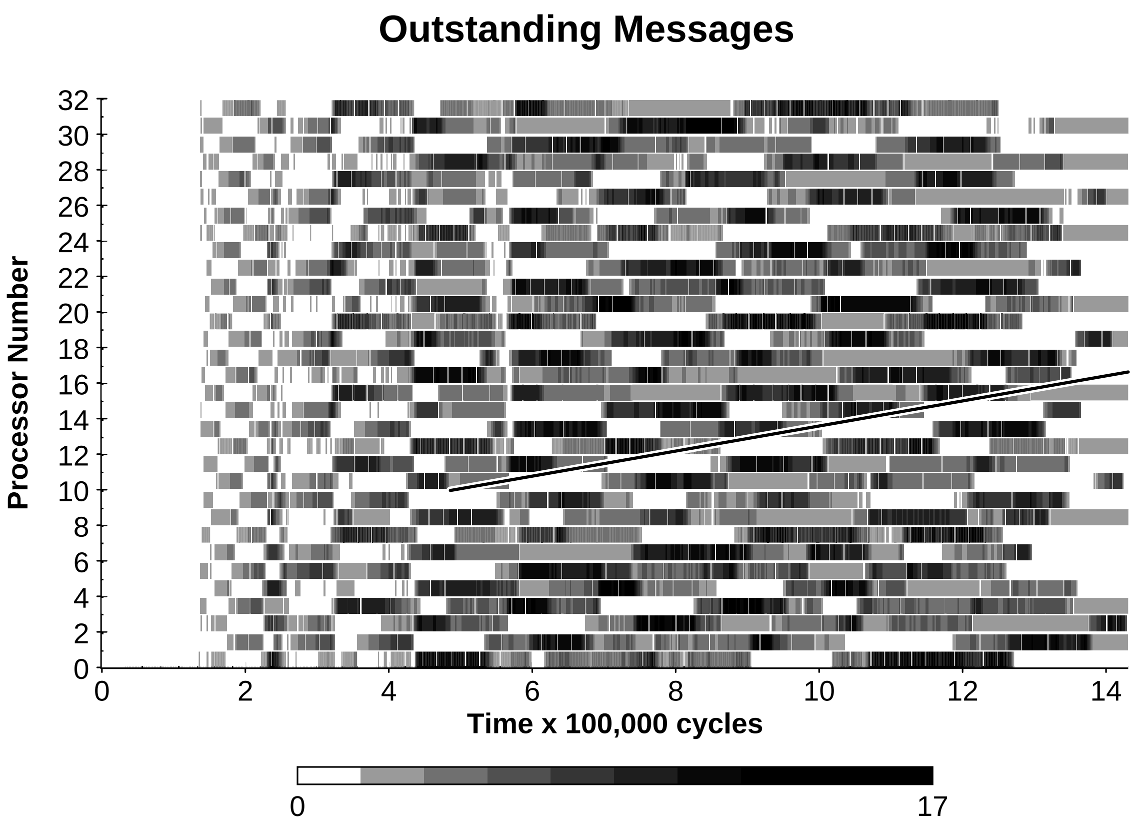


Figure 3-4: The total number of packets in the network headed to any given processor at any given time. Time is measured in 33-megahertz clock cycles. The diagonal line superimposed on the drawing follows the pattern of systematic interference caused by target collisions.

which delays the injection of packets and thus exacerbates the problem. In short order, processors alternate between being overloaded (gray) and idle (white). By 400,000 cycles, some processors have as many as 17 packets queued up. The processors above the heavily loaded processors are nearly always idle and thus appear white. These processors are idle because several senders are blocked sending to their predecessor in the shift. The white regions thus change to black in about 100,000 cycles as the group of senders transition together. These black and white regions thus form "lines" that rise at an angle of about 20 degrees from horizontal. We have explicitly marked one of these lines.

Consecutive transfers incur collisions that do not occur when the transfers are isolated with barriers. Hot spots start due to random variation and then persist systematically, getting worse and worse. The barriers increase the performance by eliminating target collisions.

We have explained, at least partly, the large differences between the experiments with barriers and the experiments without barriers. Now let us examine the other interesting features of Figure 3-3.

For the barrier-free codes, we expect the performance to drop monotonically as the block size increases, but for the largest block sizes, the performance increases unexpectedly. This is because for large block sizes, we end up doing very few cyclic shifts. For example, for a block size of 40,000 bytes, there are only $1280K/40K = 32$ different cyclic shifts. With so few transitions from one round to the next, the system never gets a chance to get as far out of sync, and the number of target collisions remains low. To demonstrate that large blocks suffer as much as medium-sized blocks, we tried running the Strata version without barriers for 100 shifts instead of 32, transferring about 3 times as much data (see Figure 3-5.) The new data point, 1.67 megabytes per second, fits right on the asymptote implied by the first half of the "Strata without Barriers" curve. Thus, without barriers, as the block size increases, the performance approaches 1.67 megabytes per second for Strata. The asymptote for CMMD is 1.43 megabytes per second.

Transfers	**MB/sec**	**95% CI**
32	2.22	± 0.122
100	1.67	± 0.0404

Figure 3-5: Big blocks also suffer from target collisions. After enough transfers without a barrier, the achieved bandwidth drops to match the asymptote of the "Strata without Barriers" curve.

Theperformance of Strata with barriers drops slightly for very large transfers, which is due to cache effects. In our experiment, each sender sends the same block over and over. For all but the largest blocks, the entire block fits in the cache. The performance of CMMD with barriers does not appear to drop with the large blocks. In actuality, it does drop, but the effects are masked by the high variance of `CMAML_scopy`. The differences in performance and variance between Strata and CMMD are quite substantial. They are due to bandwidth matching and are discussed in Section 3.6.

The versions with barriers perform worse for small blocks simply because of the overhead of the barrier, which is significant for very small transfers. The "Optimized Barriers" curve shows an optimized version that uses fewer barriers for small transfers. The idea is to use a barrier every n transfers for small blocks, where n times the block size is relatively large, so that the barrier overhead is insignificant. The actual n used for blocks of size B is $n = \lceil 512/B \rceil$. The choice of 512, which is relatively unimportant, limits barriers to about 1 for every 512 bytes transferred. Small limits add unneeded overhead, while large limits allow too many transitions between barriers and risk congestion. (At $n = \infty$, there are no barriers at all.)

Another important feature of Figure 3-3 is that the barrier-free versions have a hump for 16-byte to 400-byte block sizes. The increase from 16 to 32 bytes is due to better amortization of the fixed startup overhead of a block transfer. The real issue is why medium-sized blocks perform better than large blocks. Medium-sized blocks perform better because they do not transfer many packets to the same destination. The packet count ranges from 2 to 8 for blocks of 32 to 128 bytes. With such low packet counts, the processor switches targets before the network can back up. (Recall that the network can hold about eleven packets per node.) Between receiving and switching targets, the time for a transition allows the network to catch up, thus largely preventing the "sender groups" that form with longer block transfers. The next section examines this effect in more detail.

Finally, we have seen some cases in which using barriers more frequently than just between rounds can improve performance. This phenomenon occurs because barriers act as a form of global flow control, limiting the injection rate of processors that get ahead. Related to this, Section 3.6 shows that artificially limiting the injection rate can improve performance. We do not yet understand exactly what the additional benefit of extra barriers is over that of limiting the injection rate directly.

3.5 Packets Should Be Reordered

After the selective use of synchronization, the most important technique for maximizing bandwidth is to randomize or interleave the packets. The previous section showed that large increases in

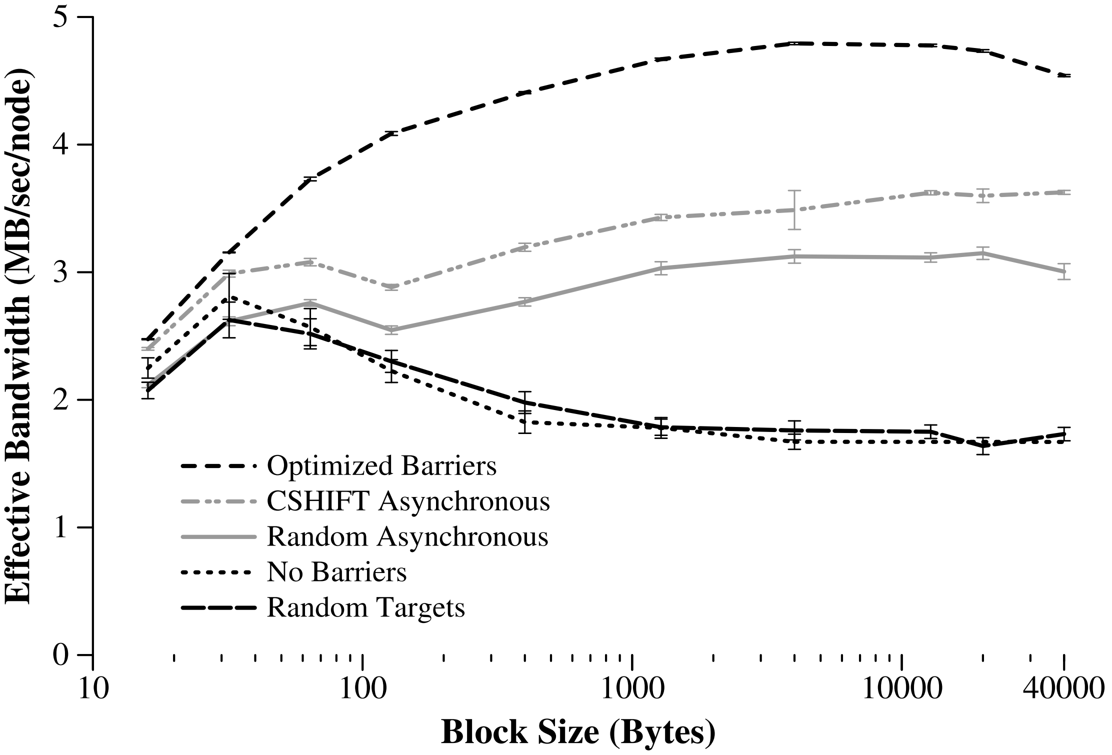


Figure 3-6: The effect of interleaving. The two asynchronous block transfer versions use packet interleaving to achieve about twice as much bandwidth as the corresponding normal block transfers. The version with barriers still performs much better, but it applies only when the communication can be structured as a sequence of permutations. The asynchronous block-transfer interface avoids this requirement.

bandwidth could be achieved by using barriers to prevent interference between adjacent rounds. When a round is not a permutation, however, collisions occur within the round and the benefit of synchronization is minimal. In this section, we show that in the cases where collisions may occur within a round but the distribution of targets is still uniform, reordering packets is the key to performance.

Figure 3-6 shows the effective bandwidth versus the size of the block sent to each target. The “Strata Block Transfer with Random Targets” version picks each new target randomly from a uniform distribution. Thus, within a round we expect target collisions and barriers not to help. The key is that for small blocks the collisions do not matter, because they are short lived. For large blocks the hot spots persist for a long time and thus back up the network, reaching the same asymptote as the cyclic-shift pattern without barriers, which also has a uniform distribution.

The key conclusion from this is that when the distribution is unknown, small batch sizes avoid prolonged hot spots. For example, if a node has ten buffers that require 100 packets each, it is much better to switch buffers on every injection (batch size of one) than to send the buffers in order (batch size of 100).

To explore this hypothesis, we built an asynchronous block-transfer interface. Each call to the asynchronous block-transfer procedure sends a small part of the transfer and queues up the rest for later. After the application has initiated several transfers, it calls a second procedure that sends all of the queued messages. The second procedure sends two packets from each queued transfer, and continues round-robin until all of the transfers are complete. To avoid systematic congestion, the order of the interleaving is a pseudo random permutation of the pending transfers. Thus, each processor sends in a fixed order, but different processors use different orders.

The performance of this technique appears in Figure 3-6 as the two "Strata Asynchronous Block Transfer" lines. For random targets, interleaving the packets increases the bandwidth by a factor of about 1.8 for large blocks (more than 400 bytes) and performs about the same for small transfers. When used for the all-pairs cyclic-shift pattern, interleaving the packets increases the performance by a factor of 2.1 for large transfers and by about 15% for small blocks. The cyclic-shift pattern performs better, because the distribution is more uniform than random targets. The version with barriers still performs substantially better, but global scheduling is required to ensure that each round is a permutation. Thus, packet interleaving should be used when the exact distribution of targets is unknown. The difference in performance between the asynchronous transfers and the version with barriers is due to the overhead for interleaving.

The dip at 128-byte transfers occurs because there is no congestion for smaller messages and because the substantial overhead of the interface is amortized for larger messages.

Packet interleaving allows the system to avoid "head-of-line" blocking, which occurs when packets are unnecessarily blocked due to the packet at the head of the queue waiting for resources that those behind it do not need. Karol *et al.* showed that head-of-line blocking can limit throughput severely in ATM networks [KHM87]. Although Strata tries to send two packets from each message at a time, if it fails, it simply moves on to the next message. This injection strategy has no effect on the CM-5, however, because the network interface contains a FIFO queue internally, which allows head-of-line blocking to occur regardless of the injection order. In general, all levels of the system should avoid head-of-line blocking.

The benefit of interleaving has important consequences for message-passing libraries. In particular, any interface in which the the library sends large one buffer at a time is fundamentally broken. Such an interface prevents interleaving. Unfortunately, the one-buffer-at-a-time interface is standard for message-passing systems. To maximize performance, a library should allow the application to provide many buffers simultaneously. The Strata interface seems quite robust, and it works best with at least four transfers at a time.

3.6 Bandwidth Matching

Given that the overhead of receiving messages limits the effective bandwidth of the network, there is no point in injecting packets any faster than the receive rate. In this section, we show that artificially limiting the injection rate improves throughput and reduces the variance in effective bandwidth of bulk data movement.

Suppose we have a message pattern in which every node both sends and receives. The ideal situation for maximizing throughput occurs when every node alternates between injection and reception. Furthermore, we would like the network to contain as few packets as possible, yet still ensure that each node always has a packet ready to be received.

Because we use polling, we can limit reception to at most one per send, unless the send fails, in which case we must poll to prevent deadlock. Unfortunately, this strategy achieves only about 2 megabytes per second, because the network becomes very congested. Thus, CMMD and Strata always choose reception over injection. They poll until the network is empty.

Although this strategy performs much better than limited polling, it is fundamentally unfair. Nodes that get a little behind may never catch up until others finish sending. The key problem is

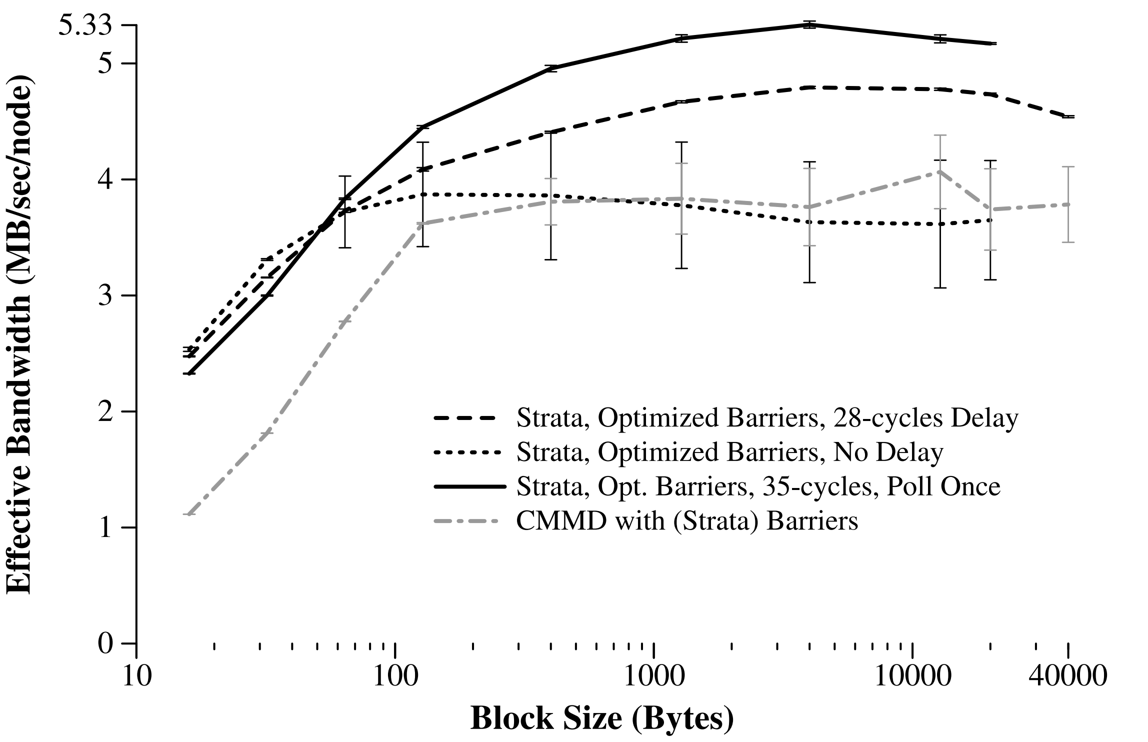


Figure 3-7: The effect of bandwidth matching on permutations separated by barriers. The error bars show 95% confidence intervals. All of the versions except "Poll Once" poll the network until it is empty. The delay value is how long the sender waits in the case that no packet arrived.

that nodes that are sending and have no pending arrivals inject packets faster than the receiver can pull them out. Furthermore, because overloaded targets are not sending, other nodes are likely to have no pending arrivals, which exacerbates the problem.

Our solution is to delay injection artificially in the case that there are no pending arrivals. This ensures that receivers pull out packets at least as fast as they arrive, and eventually the network empties. Thus, an overloaded receiver quickly catches up and resumes sending. Because we are artificially limiting the injection rate based on the expected throughput, we call this technique *bandwidth matching*.[6]

Figure 3-7 shows the impact of bandwidth matching on the cyclic-shift pattern with barriers. Without any delay, Strata actually performs worse than CMMD, because Strata has lower overhead and thus a correspondingly higher injection rate than `CMAML_scopy`. Increasing the delay to 28 cycles ensures that the injection rate is slightly slower than the reception rate. The sending overhead becomes $37 + 28 = 65$ cycles, while the receiving overhead remains at 62 cycles.

The added delay not only increases the performance by about 25%, it also reduces the standard deviation by about a factor of 50. The drop in variance occurs because the system is self-synchronizing: any node that gets behind quickly catches up and resumes sending.

[6]Bandwidth matching was discovered by Robert D. Blumofe and Eric A. Brewer when they were trying to understand why an early version of Strata's bulk transfer routines ran slower than those of CMMD. Blumofe had disassembled the CMMD code to verify that verified that Strata was programmed to perform the identical sequence of operations on the network interface as CMMD. But the Strata code was tighter, and executed fewer instructions. Brewer made the key observation, stating "It is faster because it is slower." [BB94a].

The poll-once version takes this strategy a step farther. Given that everyone is sending at nearly the same rate, it is now sufficient to pull out only one packet, since it is unlikely that there will be two packets pending. Polling when there is no arrival wastes 7 cycles. This improvement accounts for about a 7% gain in throughput. The actual reduction in message routing time is closer to 10%, however due to the fact that all nodes run in lock step, which ensures that all nodes finish at nearly the same time. Unlike the case without bandwidth matching, the network remains uncongested even though less polling occurs. Optimum performance requires both bandwidth matching and limited polling. Note that Strata sustains more bandwidth *for all-pairs* than Kwan *et al.* saw for individual messages, 10.66 versus 10.4 megabytes per second [KTR93]. The net improvement over CMMD without barriers is about 390%.

Although limited polling can improve performance, it is not very robust. When other cuts of the network, such as the bisection, become bottlenecks, limited polling causes congestion. We expect that limited polling is appropriate exactly when the variance of the arrival rate is low. If the arrival rate is bursty (due to congestion), the receiver should expect to pull out more than one packet between sends. In practice, patterns such as 2D-stencil [MS91] that do not stress the bisection bandwidth can exploit limited polling, while random or unknown patterns should poll until there are no pending packets.

Introducing delay for short transfers actually hurts performance, as shown by the superior performance of the "No Delay" version for small transfers. In this case, the startup overhead introduces plenty of delay by itself, and any additional delay simply reduces the bandwidth. Thus, future versions of Strata will adjust the delay depending on the length of the transfer. This form of adaptive delay should also remove the performance dip that appeared in the asynchronous block-transfer curves. For the limited polling case, it was beneficial to increase the delay slightly to 35 cycles to ensure that the bisection bandwidth did not affect the arrival rate.

The technique of adding delay is essentially a static form of flow control. Traditional flow control via end-to-end acknowledgments would be more robust, but very expensive, since each acknowledgment requires overhead at both ends. A relatively cheap solution for many situations is to use barriers as all-pairs end-to-end flow control. In some early experiments we found that frequent barriers improved performance. Some of this effect occurred because barriers limit the injection rate, for which bandwidth matching is more effective. We expect that frequent barriers are a more robust form of flow control, because they are a closed-loop system. Despite its lack of feedback, however, bandwidth matching is quite stable due to its self-synchronizing behavior. Finally, there has also been some theoretical evidence that introducing delays might improve performance [RU92, GL89].

3.7 Programming Rules of Thumb

The following rules-of-thumb can help programmers decide when and how to use each of these mechanisms.

- If possible, recast the communication operation into a series of permutations. Separate the permutations by barriers, and use a bandwidth-matched transfer routine, such as is provided by Strata, to implement each permutation. We found that this strategy can improve performance by up to 390%.

- If bandwidth matching is impractical, because, for example, the real bottleneck is some internal cut of the network, then using periodic barriers inside each permutation may help. We have seen cases where barriers within a permutation improve performance.

- If you know nothing about the communication pattern, you should try to arrange the communication into a bulk data transfer, and then use an interleaved or randomized injection order, as provided by Strata's asynchronous block-transfer mechanism. Even in this case, periodic barriers within the transfers may improve performance.

- It is important to keep the network empty. It is almost always better to make progress on receiving than on sending. The one exception occurs when the variance of the arrival rate is near zero (due to bandwidth matching), in which case any additional polling wastes cycles.

- If your computation operation consists of two operations, each of which has good performance separately, then keep them separate with a barrier. It is difficult to overlap communication and computation on the CM-5, because the processor must manipulate every packet, and the low capacity and message latency of the CM-5 network reduce the potential gain from such overlap. Large block transfers interact poorly with the cache, however, and we have seen cases where limited interleaving of the communication and computation can improve communications performance by about 5%.

Chapter 4

The StarTech Massively Parallel Chess Program

4.1 Introduction

After helping to design the Connection Machine CM-5, which provides hardware support for global synchronization, I wanted to explore how to exploit the MIMD characteristics of the machine fully. I looked for an application that did not fit well into the data-parallel approach, and I hit upon computer chess. The parallelism in computer chess derives from a dynamic expansion of a highly irregular game-tree, making computer chess difficult to express as a data-parallel program. To investigate how to program this sort of dynamic MIMD-style application, I engineered a parallel chess program, Startech. StarTech is based on H. Berliner's serial Hitech program [BE89], and it runs on a CM-5. This chapter explains how the StarTech program works.

A chess program searches a game tree to determine its best move. Evaluating a chess position consists of starting at the position and following the tree of possible moves. At each vertex of the tree, the program considers the set of possible moves of one of the players. The program must decide when to stop searching and once stopped, it must evaluate the position at the leaf of the tree. Both of these decisions require knowledge of chess, while the mechanism to unfold and prune the tree in a serial or parallel implementation is largely independent of the detailed chess knowledge.

The strength of a chess program depends on many factors, which we can roughly lump into two categories: chess knowledge and brute force. The chess knowledge of StarTech — which includes the opening book of precomputed moves at the beginning of the game, the endgame databases, the static position-evaluation function, and the time-control strategy — is based on H. Berliner's Hitech [BE89] program. The Hitech program runs on special purpose hardware built in the mid 1980's and searches in the range of 100,000 to 200,000 positions per second. Berliner provided us with most of an implementation of Hitech written in C that runs at 2,000 to 5,000 positions per second.[1]

The brute-force part of StarTech's chess strength derives from my parallel implementation of game-tree search. I started with a relatively crude serial alpha-beta search routine that does not perform many of the sophisticated search extensions (which change the shape of the tree being

[1]The variation in the nodes-per-second rate of our serial Hitech implementation is due to the fact that move generation is more expensive in some situations than others, and the fact that our static evaluator takes different amounts of time depending on how close the value of the chess position is to the α-β search window as used in Section 4.3.

searched)[2] found in Hitech and other modern chess programs. The parallel version of StarTech searches exactly the same tree, using the exactly the same search extensions and producing the exact same evaluations as my serial version produces when searching the same position to the same nominal search depth.

Given that the shape of the entire unpruned chess tree is fixed by the chess knowledge, the problem addressed in this chapter is to search the tree quickly and efficiently. My approach is to break the problem into two parts, an algorithm and a scheduler. The algorithm specifies *what* can be done in parallel. The scheduler specifies *when* and *on which processor* work is actually performed.

I developed a game-tree search algorithm called *Jamboree* search. The basic idea behind Jamboree search is to do the following operations on a position in the game tree that has k children:

- The value of the first child of the position is determined (by a recursive call to the search algorithm.)
- Then, in parallel, all of the remaining $k-1$ children are tested to verify that they are not better alternatives than the first child.
- Any children that turn out to be better than the first child are sequentially searched to determine which is the best.

If the move ordering is best-first, i.e., the first move considered is always better than the other moves, then all of the tests succeed, and the position is evaluated quickly and efficiently. We expect that the tests will usually succeed, because the move ordering is often best-first due the the application of several chess-specific move-ordering heuristics.

This approach to parallel search is quite natural, and variants of it have been used by several other parallel chess programs, such as Cray Blitz [HSN89] and Zugzwang [FMM91]. Still others have proposed or analyzed variations of this style of game tree search [ABD82, MC82, Fis84, Hsu90]. I do not claim that Jamboree search is new search algorithm, although some of the details of my algorithm are a bit different from the details of the related algorithms that have been previously described in the literature. Instead, I view the algorithm as a good testbed for understanding how to design scalable, predictable, dynamic MIMD-style programs.

Other parallel algorithms based on Scout search include minimal tree search, mandatory work first, and principal variation splitting. S. Akl, D. Barnard and R. Doran [ABD82] proposed the *minimal tree search*, which performs the weak α-β search by searching the minimal tree (i.e., the Knuth-Moore critical tree [KM75], which is further described in Section 6.5). Each position is kept in an expanded form for potentially a long time, resulting in unrealistic storage requirements. The Deep Thought parallel algorithm as described in Hsu's thesis [Hsu90] is a variant of the high-storage-requirement minimal tree search.

J. Fishburn [Fis84] proposed the *mandatory work first* (MWF) algorithm. Algorithm MWF is based on the weak version of α-β search. It explicitly computes the number of *critical children* of the position being searched. A child of a position is *critical* if the child is in the Knuth-Moore critical tree, which means that the child would definitely be searched by the α-β algorithm. If the position being searched has more than one critical child, then MWF searches the first child and then searches the other children in parallel. If the first child turns out to be worse than some other child, MWF then researches the children that might be the best, all in parallel — this is in contrast to

[2]StarTech's search extensions include only quiescence search with check-extension. StarTech does not use the null-move search, a decision, I have been told by computer chess experts, that costs us about a factor of two in performance. I use the same search extensions in my serial and the parallel implementations.

Jamboree which does the researches sequentially. For nodes with exactly one critical child, MWF just searches the first child. Fishburn analyzed MWF for best-ordered and worst-ordered trees, but not for realistic game trees. One can construct game trees that are mostly best-ordered, in which the MWF algorithm does almost as badly as the naive parallel α-β search's $O(\sqrt{P})$ speedup.

Fishburn's MWF algorithm can be viewed as being separate from the scheduler, but his analysis depends on the scheduler. For example, Fishburn proves that worst-ordered game-trees achieve speedup using mandatory-work-first on a tree-of-processors scheduler, in which the depth of the game-tree is much greater than the depth of the processor tree. My result, in contrast, states that the average available parallelism is less than 3, and that the speedup is less than one, for an infinite processor perfect scheduler. Even though the MWF algorithm is tangled up with the tree-of-processors scheduler one can interpret Fishburn's results someone independently of the scheduler. Fishburn's results indicate, for example, that if one has a tree of processors that is half as deep as the game tree, and the degree of the processor tree is greater than the degree of the game tree, then the critical path is short and the work efficiency is good. Such a tree is as good as "infinite processors" for an algorithm in which the shallowest $h/2$ plies of the game tree are searched in parallel and the deepest $h/2$ plies of the game tree are searched serially. It turns out that the half-the-depth-serially strategy, when applied to Jamboree search, reduces the average available parallelism even further, down to about 2 for worst-ordered trees. Fishburn did not analyze what happens if the tree of processors is as deep as the game tree. The reason that MWF achieves speedup on worst-ordered trees is that MWF researches the children who failed their tests in parallel, while the Jamboree algorithm sequentially researches all the failed children. Hence, for worst ordered trees, Jamboree search finds little parallelism, while MWF finds much parallelism. Any chess program that is searching worst-ordered trees is not competitive, however.

Several programs use principal variant splitting (PV-splitting) [MC82], which is a another variation on MWF, but the ideas behind PV-splitting are, like MWF, somewhat obscured by the fact that a tree-of-processors scheduler is entangled into the search algorithm. Later work has separated the scheduler from the algorithm; For example, Cray Blitz [HSN89] apparently uses PV-splitting with something like a work-stealing scheduler. No critical path analysis or measurement has been performed for

The Zugzwang program, developed by R. Feldmann, P. Mysliwietz, and B. Monien [FMM91], uses a parallel search algorithm that is very similar to Jamboree search. Zugzwang achieves high work-efficiency, searching to within a few percent the same number of nodes in a parallel search as in a sequential search. The efficiency of StarTech appears to be somewhat lower, probably because the Zugzwang team has gone to substantial effort to try to ensure that they search the tree in a mostly best-first order.

The parallel aspiration search algorithm [Bau78] divides the α-β window into segments, and gives each processor a different segment of the window to search. Aspiration search achieves only small parallel speedups. Surprisingly, the serial version of aspiration search often runs faster than a infinite window search. Today most state-of-the-art chess programs, including StarTech, use a serial aspiration search in which the game tree is searched with a small α-β window, and if the score is outside of the window, the tree is researched.

R. Karp and Y. Zhang [KZ89] show how to search an AND/OR tree in parallel by carefully allocating the right number of processors to each subtree. C. Stein [Ste92] employs Karp and Zhang's algorithm as a subroutine to do a parallel α-β search. Stein performs a binary search for the value of the game tree, at each stage converting the game tree to an AND/OR tree with the question "Is the value of the root greater than s?".

There are several other approaches to game tree search that are not based on α-β search. H. Berliner's B* search algorithm [Ber79] tries to prove that one of the moves is better according to a pessimistic evaluation than any of the other moves are according to a optimistic evaluation. D. McAllester's Conspiracy search [McA88] expands the tree in such a way that to change the value of the root will require changing the values of many of the leaves of the tree. The SSS* algorithm [Sto79] applies branch and bound techniques to game tree search. These algorithms all require space which is nearly proportional to the run time of the algorithm, but the the constant of proportionality may be small enough to be feasible. While these algorithms all appear to be parallelizable, they have not yet been successfully demonstrated as practical serial algorithms. I wanted to be able to compare my work to the best serial algorithms.

To distribute work among the CM-5 processors, StarTech uses a work-stealing approach, in which idle processors request work. Processors run code that is nearly serial. When a processor discovers some work that could be done in parallel, it *posts* the work into a local data structure. When a processor needs work, it sends a message to another processor, selected at random, and removes work from that processor's collection of posted work. StarTech uses a globally synchronous throttling mechanism to prevent idle processors from swamping busy processors with requests for work.

Many researchers have tried to build parallel chess programs, with mixed success. The StarTech program owes its success both to good hardware and good software. On the hardware side, the CM-5's fast user-level message passing capability makes it possible to use a global transposition table, and to distribute fine grained work efficiently. The CM-5 control control network makes it easy to avoid swamping problems. Fast timing facilities allow fine-scale performance measurement. On the software side, StarTech uses a good search algorithm, and systematically measures critical path length and total work to understand the performance of the program.

This chapter explains how StarTech works. First Section 4.2 briefly reviews how unpruned game tree search can used to evaluate a chess position, while Section 4.3 reviews α-β pruning, and Section 4.4 reviews Pearl's Scout search algorithm. Section 4.5 describes the Jamboree search algorithm. Then Section 4.6 describes some of the general problems of scheduling dynamic tree searches on MIMD machines. Finally, Section 4.7 explains how the StarTech scheduler works.

4.2 Negamax Search Without Pruning

Before delving into the details of the Jamboree algorithm, let us review the basic search algorithms that are applicable to computer chess. Most chess programs use some variant of negamax tree search to evaluate a chess position. The goal of the negamax tree search is to compute the value of position p in a tree T_p rooted at position p. The value of p is defined according to the negamax formula:

$$v_p = \begin{cases} \texttt{static\_eval}(p) & \text{if } p \text{ is a leaf in } T_p \text{, and} \\ \max\{-v_c : c \text{ a child of } p \text{ in } T_p\} & \text{if } p \text{ is not a leaf.} \end{cases}$$

The negamax formula states that the best move for player A is the move that gives player B, who plays the best move from B's point of view, the worst option. If there are no moves, then we use a static evaluation function. Of course, no chess program searches the entire game tree. Instead some limited game tree is searched using an imperfect static evaluation function. Thus, we have formalized the chess knowledge as T_p, which tells us what tree to search, and `static_eval`, which tells us how to evaluate a leaf position.

The naive Algorithm `negamax` shown in Figure 4-1 computes the negamax value v_p of position p by searching the entire tree rooted at p. It is easy to make Algorithm `negamax` into a parallel algorithm, because there are no dependencies between iterations of the *for* loop of Line (N5).

One simply changes the *for* loop into a parallel loop. But negamax is not a efficient serial search algorithm, and thus, it makes little sense to parallelize it.

Define `negamax`(p) as
  If n is a leaf then return `static_eval`(n).
  Let $\vec{c} \leftarrow$ the children of n, and
    $b \leftarrow -\infty$.
    For i from 0 below $|\vec{c}|$ do:
      Let $s \leftarrow -$`negamax`$(\vec{c}_i)$. *;; Recursive Search*
        if $s > b$ then set $b \leftarrow s$. *;; New best score*
     enddo
    return b.

Figure 4-1: Algorithm `negamax`.

4.3 Alpha-Beta Pruning

The most efficient serial algorithms for game-tree search all avoid searching the entire tree by proving that certain subtrees need not be examined. In this section we review the α-β serial search algorithm in preparation for the explanation of how the Jamboree parallel search algorithm works.

An example of how pruning can reduce the size of a game tree that is searched can be seen in the chess position of Figure 4-2. Suppose White has determined that it can win Black's queen with 40. ♔×h2. White's other legal move 40. ♔f1 fails to capture the queen. White does not need to consider every possible way for Black's queen to escape. Any one of a number of possibilities suffices. Thus, white can stop thinking about the move without having exhaustively searched all of Black's options.

The idea of pruning subtrees that do not need to be searched is embodied in the serial α-β search algorithm [KM75], which computes the negamax score for a node without actually looking at the entire search tree. The algorithm is expressed as a recursive subroutine with two new parameters α and β. If the value of any child, when negated, is as great as β, then the value of the parent is no less than β, and we say that the parent *fails high*. If the values of all of the children, when negated, are less than or equal to α, then the value of the parent is no greater than α, and we say that the parent *fails low*.

Procedure `absearch`[3] is shown in Figure 4-3. When Procedure `absearch` is called, the parameters α and β are chosen so that if the value of a node is not greater than α and less than β, then we know that the value of the node can not affect the negamax value of the root of the entire search tree. After the score is returned from the subsearch on Line (A6), the algorithm, on Line (A7), checks to see if the negated score is as great as β. If so, we know that the value of the node is at least as great as β and we can skip searching the remaining children — The procedure has *failed high* in this situation. Just because one of the children has a negated score less than α,

[3]This variant on the standard α-β algorithm is due to Fishburn [Fis83], who called it *fail-soft α-β search.* Fail-soft α-β search can return a value that is less than α, in which case the value returned is an upper bound to the true value of the node, or the search can return a value that is greater than β, in which case the value returned is a lower bound to the true value.

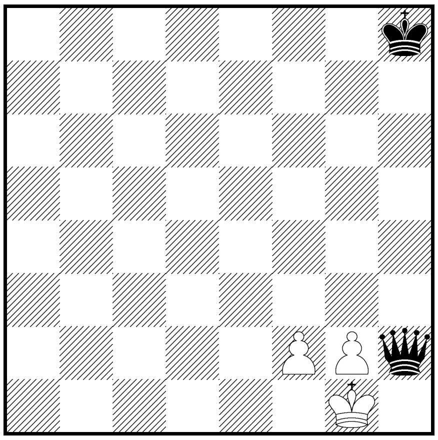


Figure 4-2: White to move and win. In this position, White need not consider all of Black's alternatives to 40. ♔f1, since almost any move Black makes will keep the queen, a worse outcome than just taking the queen with 40. ♔×h2.

however, does not mean that some other child might not be within the α-β window. The algorithm can only *fail low* after considering all of the children.

```
Define absearch(n, α, β) as
    If n is a leaf then return static_eval(n).
    Let  c⃗ ← the children of n, and
         b ← −∞.
       For i from 0 below |c⃗| do:
          Let s ← −absearch(c⃗_i, −β, −α).
             If s ≥ β then return s.              ;; Fail High
             If s > α then set α ← s.             ;; Raise α
             If s > b then set b ← s.
        enddo
       return b.
```

Figure 4-3: Algorithm absearch.

The α-β algorithm can substantially reduce the size of the tree searched. The α-β algorithm works best if the best moves are considered first, because if any move can make the position fail high, then certainly the best move can make the position fail high. Knuth and Moore [KM75] show that for searches of a uniform best-ordered tree of height H and degree D, the α-β algorithm searches only $O(\sqrt{D^H})$ leaves instead of D^H leaves.

For any $k \geq 0$, before searching the $(k+1)$st child, the α-β algorithm obtains the value of the kth child and possibly uses that value to adjust α or return immediately. This dependency between finishing the kth child and starting the $(k+1)$st child completely serializes the α-β search algorithm.[4]

[4] R. Finkel and J. Fishburn showed that if the serialization implied by α-β pruning is ignored by a parallel program,

4.4 Scout Search

For a parallel chess program, we need an algorithm that both effectively prunes the tree and can be parallelized. We started with a variant on serial α-β search, called *Scout* search, and modified it to be a parallel algorithm. This section explains the Scout search algorithm.

The serial Scout search algorithm, due to J. Pearl [Pea80], is shown in Figure 4-4. Procedure `scout` is similar to Procedure `absearch`, except that when considering any child that is not the first child, a *test* is first performed to determine if the child is no better a move than the best move seen so far. If the child is no better, the test is said to *succeed.* If the child is determined to be better than the best move so far, the test is said to *fail,* and the child is searched again *(valued)* to determine its true value.

The Scout algorithm performs tests on positions to see if they are greater than or less than a given value. A test is performed by using an empty-window search on a position. For integer scores one uses the values $(-\alpha - 1)$ and α as the parameters of the recursive search, as shown on Line (S9). A child is tested to see if it is worse than the best move so far, and if the test fails on Line (S12) (i.e., the move looks like it might be better than the best move seen so far), then the child is valued, on Line (S13), using a non-empty window to determine its true value.

(S1) Define $\texttt{scout}(n, \alpha, \beta)$ as
(S2)   If n is a leaf then return $\texttt{static\_eval}(n)$.
(S3)   Let $\vec{c} \leftarrow$ the children of n, and
(S4) $b \leftarrow -\texttt{scout}(c_0, -\beta, -\alpha)$.
(S5)   *;; The first child's valuation may cause this node to fail high.*
(S6)   If $b \geq \beta$ then return b.
(S7)   If $b > \alpha$ then set $\alpha \leftarrow b$.
(S8)   For i from 1 below $|\vec{c}|$ do: *;; the rest of the children*
(S9)   Let $s \leftarrow -\texttt{scout}(\vec{c}_i, -\alpha - 1, -\alpha)$.   *;; Test*
(S10)   If $s > b$ then set $b \leftarrow s$.
(S11)   If $s \geq \beta$ then return s.   *;; Fail High*
(S12)   If $s > \alpha$ then   *;; Test failed*
(S13)   Set $s \leftarrow -\texttt{scout}(\vec{c}_i, -\beta, -\alpha)$.   *;; Research for value*
(S14)   If $s \geq \beta$ then return s.   *;; Fail High*
(S15)   If $s > \alpha$ then set $\alpha \leftarrow s$.
(S16)   If $s > b$ then set $b \leftarrow s$.
(S17)   enddo
(S18)   return b.

Figure 4-4: Algorithm `scout`.

If it happens to be the case that $\alpha + 1 = \beta$, then Line (S13) never executes because $s > \alpha$ implies $s \geq \beta$, which causes the *return* on Line (S11) to execute. Consequently, the same code for Algorithm `scout` can be used for the testing and for the valuing of a position.

Line S10, which raises the best score seen so far according to the value returned by a test, is necessary to insure that if the test fails low (i.e., if the test succeeds), then the value returned is an upper bound to the score. If a test were to return a score that is not a proper bound to its parent, then

then it will achieve only $\sqrt{P}$ speedup on P processors [FF82].

the parent might return immediately with the wrong answer when the parent performs the check of the returned score against β on Line S11.

A test is typically cheaper to execute than a valuation because the α-β window is smaller, which means that more of the tree is likely to be pruned. If the test succeeds, then algorithm `scout` has saved some work, because testing a node is cheaper than finding its exact value. If the test fails, then `scout` searches the node twice and has squandered some work. Algorithm `scout` bets that the tests will succeed often enough to outweigh the extra cost of any nodes that must be searched twice, and empirical evidence [Pea80] justify its dominance as the search algorithm of choice in modern serial chess-playing programs.

4.5 Jamboree Search

The Jamboree algorithm, shown in Figure 4-5, is parallelized version of the the Scout search algorithm. The idea is that all of the testing of the children is done in parallel, and any tests that fail are sequentially valued. A parallel loop construct, in which all of the iterations of a loop run concurrently, appears on Line (J7). Some synchronization between various iterations of the loop appears on Lines J12 and J18. We sequentialize the full-window searches for values, because, while we are willing to take a chance that an empty window search will be squandered work, we are not willing to take the chance that a full-window search (which does not prune very much) will be squandered work. Such a squandered full-window search could lead us to search the entire tree, which is much larger than the pruned tree we want to search.

(J1) Define `jamboree`(n, α, β) as
(J2)   If n is a leaf then return `static_eval`(n).
(J3)   Let $\vec{c} \leftarrow$ the children of n, and
(J4) $b \leftarrow -$`jamboree`$(c_0, -\beta, -\alpha)$.
(J5)   If $b \geq \beta$ then return b.
(J6)   If $b > \alpha$ then set $\alpha \leftarrow b$.
(J7)   In Parallel: For i from 1 below $|\vec{c}|$ do:
(J8)   Let $s \leftarrow -$`jamboree`$(\vec{c}_i, -\alpha - 1, -\alpha)$.
(J9)   If $s > b$ then set $b \leftarrow s$.
(J10)   If $s \geq \beta$ then abort-and-return s.
(J11)   If $s > \alpha$ then
(J12)   Wait for the completion of all previous iterations
(J13)   of the parallel loop.
(J14)   Set $s \leftarrow -$`jamboree`$(\vec{c}_i, -\beta, -\alpha)$. *;; Research for value*
(J15)   If $s \geq \beta$ then abort-and-return s.
(J16)   If $s > \alpha$ then set $\alpha \leftarrow s$.
(J17)   If $s > b$ then set $b \leftarrow s$.
(J18)   Note the completion of the ith iteration of the parallel loop.
(J19)   enddo
(J20)   return b.

Figure 4-5: Algorithm `jamboree`.

The *abort-and-return* statements that appear on Lines J10 and J15 return a value from Procedure `jamboree` and abort any of the children that are still running. Such an abort is needed when the

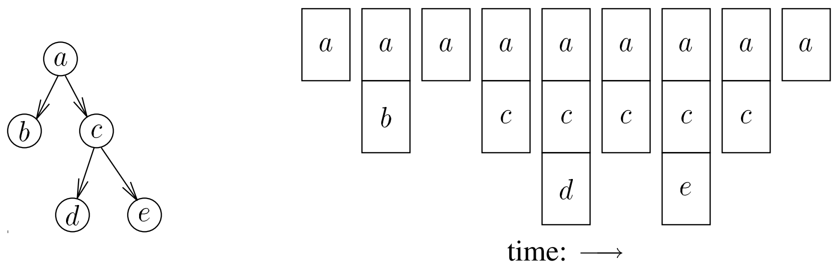


Figure 4-6: A search tree and the sequence of stack configurations for a serial implementation.

procedure has found a value that can be returned, in which case there is no advantage to allowing the procedure and its children to continue to run, using up processor and memory resources. The abort causes any children that are running in parallel to abort their children recursively, which has the effect of deallocating the entire subtree.

I have considered several variants on Jamboree search, such as to value the first two children before starting the parallel tests of all the remaining children. By valuing the first two children, the algorithm is guaranteed to get a score that prunes at least one of the children. Such a variant might have less parallelism but search fewer positions. If there is still enough parallelism, then we would prefer the more efficient algorithm.

Parallel search of game-trees is difficult because the most efficient algorithms for game-tree search are inherently serial. Good game-tree search algorithms use *pruning* to reduce the size of the tree that needs to be searched. As an extreme example consider a position which is mate-in-one, i.e., the player can immediately win by making the correct move. As soon as the program has found a way to obtain mate-in-one, it does not need to consider any of the alternative moves. It does not care that the third move leaves it three pawns ahead or even that there is more than one way to achieve mate-in-one. Thus, as soon as a good move is made, that knowledge can be used to prune the search tree without affecting the final answer. If the serial program examines the mate-in-one as its first alternative, then it will only search one node. In this example, there is no available parallelism. Trying to exploit parallelism by searching additional nodes of the tree is fruitless, since those additional nodes would never have been searched by a serial algorithm. In order to search a game-tree in parallel, we must take the risk that we may perform extra work that a good serial program would avoid.

4.6 Multithreading with Active Messages

Now that we have studied the parallel search algorithm, let us direct our attention to how one can implement a tree-searching algorithm, such as Jamboree search, on a distributed memory MIMD machine, such as the CM-5, in which all communication between processors is done with messages.

Consider the general problem of implementing recursive tree-searching programs. In a serial implementation of a recursive tree searching program, a *stack* of activation frames is used to maintain the state of the program. Each frame holds the arguments, local variables, and the program counter for the local computation. Consider the tree shown at the left in Figure 4-6, with vertices labeled a, b, c, d, and e. The stack of frames goes through the configurations at the right in Figure 4-6 (our stack grows and trees both grow downward), and at any instant only the bottommost frame is active.

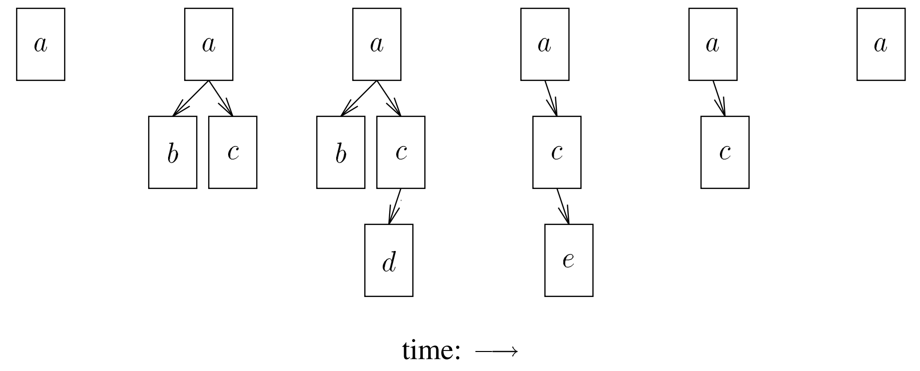


Figure 4-7: A sequence of activation trees for a parallel tree search.

In a parallel implementation, however, calls to b and c may run concurrently, and the last-in-first-out discipline for allocating activation frames does not work. A parallel implementation must use a *tree* of activation frames rather than a *stack* of activation frames.[5]

At any instant, any of the frames can be active, not just frames at the leaves of the tree. Figure 4-7 shows one way to expand the tree in parallel. In this example, child d could have been allocated in parallel with child e, but the scheduler chose to run child d and e serially. In a chess program, the game tree being searched is partially mirrored by the instantaneous state of the activation tree.

The system needs to deallocate frames. If frames were never deallocated, our space bounds would rise to meet our time bounds, and we would not be able to search very large game trees. The system cannot deallocate a frame until all of the messages destined for the frame have been received. Otherwise, a message could arrive and be interpreted by a frame that has been reallocated, and system would become corrupted. One must be careful that there are neither any other processors about to send a message to nor any messages in the network destined for a frame that is about to be deallocated.

One way to determine when all the messages have been delivered from Frame A to Frame B is, when A has decided to stop communicating to B, to send a *completion* message from A to B. If the interprocessor communications network is first-in-first-out, then this strategy suffices. When Frame B receives completion messages from every frame with which B is communicating, then B can safely deallocate itself. The CM-5 communications network, however, does not guarantee that messages are delivered in the same order that they were injected into the network.

To solve the deallocation problem, the completion message needs to contain a count of the number of messages that have been sent from A to B. When B has received all its completions and has accounted for every message described in the completion, then B can deallocate itself.

For a tree search algorithm we can optimize this communication by piggybacking the completion messages with the messages that would be otherwise sent anyway. In the Jamboree parallel search algorithm, there are three kinds of communications between frames:

[5]In StarTech, we use a tree of activation frames, but each processor has a single call stack on which the activation frames are kept. This implementation decision affects the space bounds of the program, which we will consider in Chapter 5, but is independent of the issues presented in this section. R. Nikhil and Arvind expressed clearly the need for a tree of activation frames [NA89]. Many systems have been designed to support a tree of activation frames, including dataflow machines such as EM-4 [SYH*89] and Monsoon [PC90], combinator reduction interpreters [Tur79, DR81], and multiprocessor LISP systems [Hal84], and threaded interpreters [Cla87, CSS*91, NPA92, Hal94].

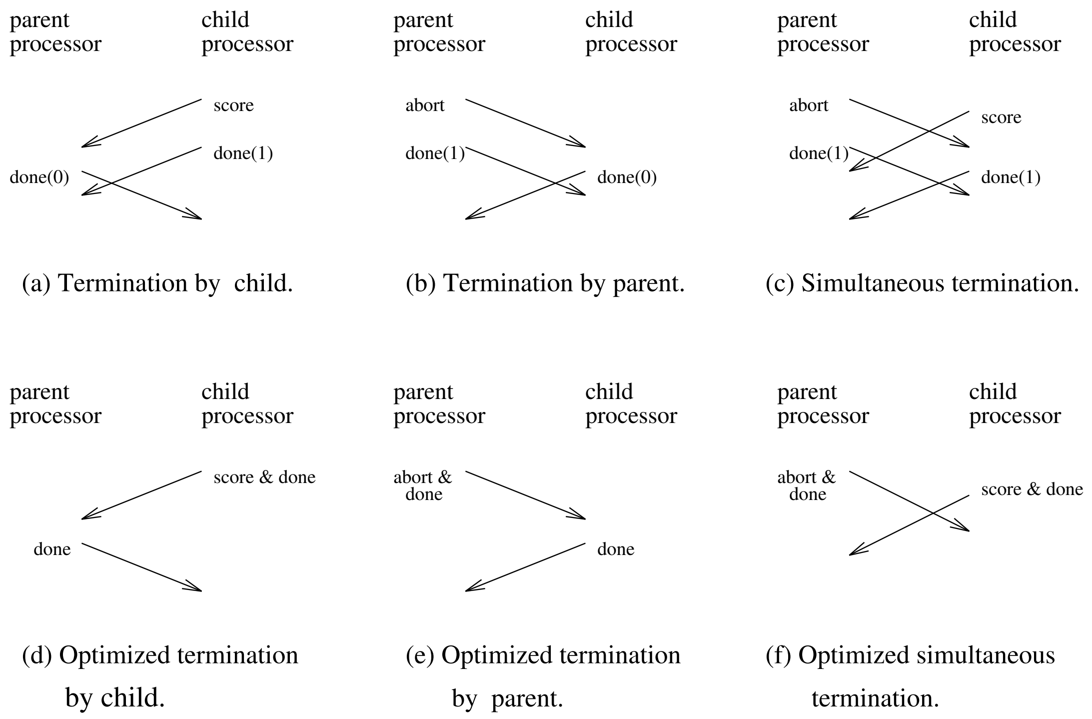


Figure 4-8: Three possible ways to for a frame to terminate (left to right). In the naive implementation (top), either a 'score' message or an 'abort' message can trigger the 'done' messages, which contain a message-count. In the optimized implementation (bottom), the 'done' message is piggybacked with the 'score' or the 'abort' message.

- When creating a new frame, messages from parent to child pass the arguments for the subsearch.
- When the value of a frame has been computed, the child sends a result its parent.
- The parent can abort one of its children by sending an 'abort' message to the child. When an abort message arrives at a frame, the frame aborts all of its children recursively.[6]

Figure 4-8 shows three possible ways for a frame to terminate. Figure 4-8(a) shows a 'normal' return, where the child returns the score to the parent, and the two frames exchange completion messages. Figure 4-8(b) shows a 'abort' scenario, where the parent tells the child to abort, then the completions are exchanged. And Figure 4-8(c) shows a situation where the parent decides to abort the child and the child decides to return a result at the same time, and then completions are exchanged.

We can optimize some of these messages away. Observe that the result is always immediately followed by a completion (but completions are not always preceded by a result) to the parent, and the abort is always immediately followed by a completion (but completions are not always preceded by an abort) to the child. The StarTech implementation merges the completion and result messages,

[6]Recall that the Jamboree search algorithm aborts its children when another child returns a score causing the node to fail-high. Thus, aborts are caused by pruning that occurs after a child has started.

and merges the completion and abort messages, as shown in Figure 4-8(d)–(f). As a consequence of using fewer messages, the protocol is simplified, and the performance is improved.

A variation of my method of removing acknowledgments from the abort/result protocol appears in distributed systems. A. Tanenbaum and R. van Rennesse show how to remove acknowledgments from remote procedure calls [Tv85]. They are not concerned about the dangling reference problem, but rather simply with reducing the number of messages.

4.7 The StarTech Scheduler

Now that we have a search algorithm and have reviewed how to run tree searches on a MIMD machine, we explain the details of the StarTech scheduler. The job of a scheduler is to decide when to execute work and on which processor to execute it. The scheduler tries to run the program quickly and without using too much memory. This section describes how work is scheduled by StarTech.

StarTech uses a work-stealing strategy, in which idle processors request work from busy processors, to schedule work. Each busy processor maintains a collection of jobs that can be stolen and worked on in parallel. These jobs can be thought of as 'frame stubs' which, in this application, contain the information needed to perform a search of some subtree in the Jamboree algorithm.

In StarTech, all interprocessor communication is performed using active messages. I used the Berkeley CMAM active message library [vCG*92], since when I started this project, the CMMD library did not support active messages. Active messages allowed me to implement many of our protocols directly and efficiently.

Each processor executes a program which is as close to the standard serial program as possible. StarTech uses the standard C-language call stack to implement all the activation frames that are scheduled onto a given processor at the same time. Thus, within one processor, the program runs nearly as efficiently as the serial version of the program, and the overhead for parallelism only is paid when work is stolen. Communication among frames on the same processor is done directly through memory rather than through active messages.[7]

A processor that needs work sends a "request-to-steal" active message to another, randomly selected processor. If the other processor has work available to steal, the requester is sent a "request-granted" message which includes a description of the work. If the other processor has nothing available to steal, then the requester is sent a "request-denied" message, and the requester tries again by sending another "request-to-steal" message to another random processor. This work stealing protocol is an example of a request-reply protocol. Section 2.2 showed how to implement deadlock-free request-reply protocols on the CM-5.

If, when a request-to-steal message arrives at a processor, there are many jobs available to be stolen, the StarTech scheduler prefers to steal jobs that are nearer the root of the tree. Since the jobs nearer the root correspond to deeper subtrees, StarTech's policy tends to steal the largest

[7] Both DIB [FM87] and Mul-T [MKH91] use the approach optimizing accesses within a single processor. DIB uses two representations for its frames: and on-processor stack and an inter-processor tree. DIB dynamically converts between the two representations. DIB was designed to run on a collection of computers operating over a local area network, and so it includes many mechanisms to cope with failing processors. The high-cost of interprocess communication on such a collection discourages message-sending, and DIB's approach to global data structures is correspondingly awkward. The Mul-T system optimizes access within a single processor by delaying the creation of closures and other run-time structures until work is actually stolen. The single-processor performance of a Mul-T program remains substantially lower than a program written in C, however.

jobs possible, which improves the chances of amortizing the overhead of stealing the work. This scheduling policy is written directly into the active message handler.[8]

One problem that can arise with a simple work-stealing scheduler is that all the idle processors (the thieves) can gang up on a busy processor, *swamping* it with requests for work, thereby preventing it from getting any work done. We now explain how StarTech handles the swamping problem. In the next chapter (in Section 5.7) we study the underlying causes of the swamping problem.

StarTech uses the CM-5 global synchronization network to help solve the swamping problem. The synchronization is used as a global *throttle* on the activity of the thieves. Using the split-phase global synchronization operation, each processor can "raise its hand", then do some more work, and then query to find out if everyone has raised its hand. The StarTech throttling rule is simple:

- Each busy processor must do some work before raising its hand.
- Each idle processor raises its hand immediately. The idle processor can send out only one request and, if the request is denied, must wait until everyone raises its hand before it can send another request.

After everyone has raised its hand, the global synchronization network resets, and everyone can raise its hand again. Meanwhile, busy processors continue to execute their work regardless of the state of the global synchronization (see Figure 4-9.) The global throttle does not stop any other busy processors from getting more work done, because the global throttle uses split-phase barriers. When a busy processor enters a barrier, it keeps working until the barrier completes.

Thus, StarTech's allocation of new activation frame proceeds in globally synchronous phases. During each phase, the expected number of request-to-steal messages arriving at any processor is less than one. The StarTech global-throttle requires very little tuning, since the slowest processor to "get some work done" automatically controls the allocation rate. The only processors that are slowed by the barrier are the idle processors. The busy processors, as shown above, are not slowed down. Thus, by tuning the frequency at which the global throttle is completed, we leave the performance of the busy processors unchanged, and only the time it takes for an idle processor to find work is affected. This tuning is only important if there is not enough parallelism to make the granularity of the work large relative to the time between globally synchronous phases. Thus, we see that fast global barriers are useful for implementing dynamic MIMD-style computation, as well as for implementing data-parallel programming.

[8] In StarTech, we use polling rather than interrupt-driven messages. With this approach atomicity can be guaranteed for a critical section of code simply by not polling the network during the critical section.

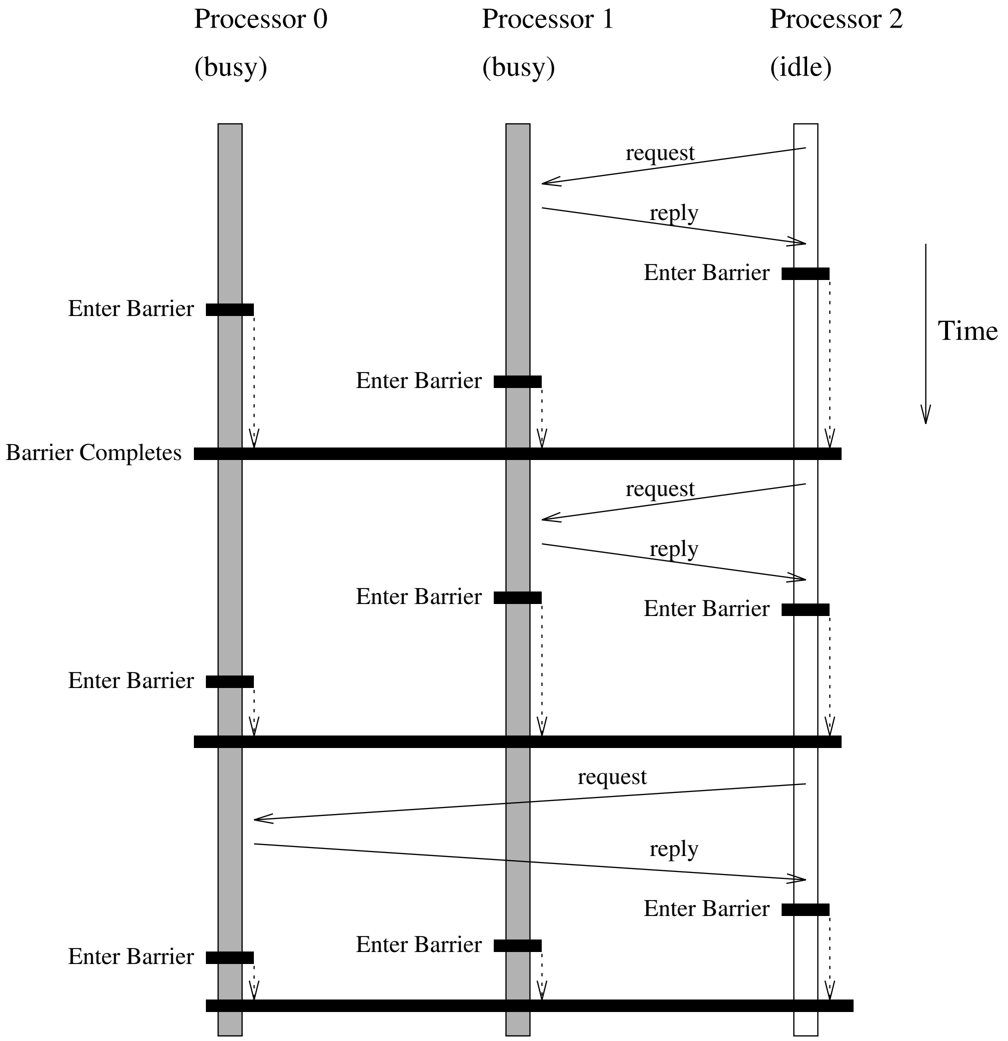


Figure 4-9: Busy processors keep working while the split-phase barrier completes. The split-phase barrier does not complete until all processors have entered the barrier. The idle processors are not allowed to send another request until the barrier completes.

Chapter 5

The Performance of the StarTech Program

5.1 Introduction

In Chapter 4 we explained how the StarTech parallel chess program works. In this chapter, we study the performance of the StarTech program, relating the performance of the program to three fundamental performance parameters: critical path length, work, and work efficiency. We shall study how to use those performance parameters to tune the StarTech program in Chapter 6. Recall that my strategy for parallel computer chess is to divide the program into two parts: the search algorithm and the scheduler. The algorithm specifies what can be done in parallel, while the scheduler decides when and where to do the work. Our performance study is organized along similar lines. We first study the critical path, total work, and work efficiency of the Jamboree game-tree search algorithm. Then, we study the StarTech scheduler, using the critical path and total work as a benchmark with which we determine how well the StarTech scheduler does compared to optimal scheduling.

My strategy for understanding the performance of StarTech is quite different from the strategy used by other researchers, such as T. Marsland, M. Olafsson and J. Schaeffer [MOS86]. Marsland *et al.* measure *overheads* to try to understand the performance of their parallel chess programs. If the serial program runs in time T_s then ideally the performance on P processors would be T_s/P. They define the *total overhead* of the program to be any additional time taken to actually run the program. Thus, if it actually takes T_P to run on P processors, the total overhead is $T_P - T_s/P$. To try to understand where the total overhead is coming from, Marsland *et al.* break the overhead into three components:

Communication overhead: The time spent sending messages.

Search overhead: The cost attributable to extra positions searched by the parallel program as compared to the number of positions searched by the serial program.

Synchronization overhead: The time processors are idle. (Referred to as synchronization overhead because any idle processor must be waiting for some other processor to do something.)

There are problems with overhead measurement, however. To measure communication overhead, Marsland *et al.* count the number of messages and multiply by the cost of sending a single message. To measure the search overhead, they count the number of positions searched. To measure the synchronization overhead, they subtract the communication overhead and the search overhead from the total overhead. Each of these measurements is wrong. For example, communications overhead depends on how congested the network is.[1] The number of positions searched does not directly relate to the amount of work performed, since move generation and static evaluation takes different amounts of time for different positions. The synchronization overhead is not measured independently, so there is no way to quantify the errors incurred by approximating the other two overheads.

Furthermore, overhead measurement does not provide much insight into the performance of the program. There is no way to distinguish how much of the synchronization overhead is incurred because the program has insufficient parallelism, and how much is incurred because the scheduler does a poor job of distributing the parallel work among the processors. By instead measuring the critical path length, the work, and the work efficiency, we can obtain precise explanations of how a program achieves its performance.

To define our three measures of parallel performance, we think of a parallel algorithm as an acyclic directed graph: dataflow graph. We define

- the *critical path length* C of the graph as the depth of the graph, measured in units of time; and
- the *work* W of the graph as the sum of the times to execute all of the instructions in the graph.

These two values, which can be derived analytically or measured empirically, can provide useful time bounds. The critical path length C of a dataflow graph gives a lower-bound to the time it takes to execute the graph. Even given an infinite number of processors and a perfect scheduler, it takes at least time C to execute the graph, because all of the instructions on any path must be executed sequentially. The work W provides another lower-bound to the run time. On P processors, one cannot hope to perform W units of work in time less than W/P. Thus, on P processors, the time T_P to execute a graph must satisfy

$$T_P \geq W/P, \text{ (the `linear speedup' lower bound),} \tag{5.1}$$

$$T_P \geq C \text{ (the `critical path' lower bound.)} \tag{5.2}$$

Brent showed [Bre74, Lemma 2] that for every dataflow graph there is a schedule that executes the graph in no more than the sum of the linear speedup term and the critical path term. That is, there are schedules such that

$$T_P \leq \frac{W}{P} + C. \tag{5.3}$$

Blumofe and Leiserson [BL93] show that any greedy schedule that has no overheads achieves Brent's bounds.

The work W and the critical path length C can be combined to give the average available parallelism of a program. If you know W and C, you can produce lower bounds on the time to run on P processors. If C is as large as W, then even with an unbounded number of processors and a perfect scheduler you won't achieve much speedup. On the other hand if C is very small compared to W, then you can conceivably make use of many processors to achieve speedup. This relationship between C and W can be captured by the average available parallelism

$$A = W/C.$$

[1] The number of messages may be a good measure of the communications overhead in a machine where sending a message requires an operating system call taking tens of thousands of instructions to execute. On the CM-5, however, where sending a message consumes only a few processor cycles, the actual performance of the network can dominate the communications overhead.

The average available parallelism tells us how many processors we can fruitfully apply to running a program. From Brent's theorem we can conclude that if $P \leq A$ then $C \leq W/P$ and therefore that $T_P \leq 2W/P$. That is if we have more average available parallelism than processors, we can, with the right scheduler, get within a factor of two of linear speedup.

Often, more work is done by a parallel algorithm than by the best serial algorithm. We define the *work efficiency* of a parallel computation to be the ratio of the parallel work of the computation to the serial work done by the best serial implementation to solve the same problem. Brent's theorem uses the amount of parallel work rather than the amount of serial work, and so even with an ideal scheduler, if the work efficiency of the algorithm is too low, then the computation will be slow.

This chapter examines the performance of the Jamboree algorithm using both analytical and empirical techniques. We analyze two special cases: best-ordered uniform game trees and worst-ordered uniform game trees. A *best-ordered* game tree is a tree in which at every position in the tree, we consider the best move first. That is, we first search the subtree that corresponds to the best move. In Chapter 4 we observed that α-β search and Scout search both behave optimally on best-ordered trees. A *worst-ordered* tree is one in which we consider the worst move first, and then the second worst move, and so on, finally considering the best move last. In such trees, α-β does not perform any pruning at all, and Scout search searches the entire tree, even visiting some positions in the tree more than once. It turns out that Jamboree search also does a very good job on best-ordered trees, but that Jamboree search offers no opportunity for speedup on worst-ordered trees.

After studying the performance of the Jamboree algorithm, we direct our attention to the StarTech scheduler. Given the critical path and total work performed by the Jamboree search algorithm, we construct a performance model for the program. I found that, on real chess positions, the StarTech scheduler achieves

$$T_P \approx 1.02\frac{W}{P} + 1.5C + 4.3 \text{ seconds.} \tag{5.4}$$

Except for the 4.3 second startup cost, the performance of the StarTech scheduler is within a factor of 2.52 of optimal.

In addition to modeling the system as a whole, we study one of the problems that commonly occur in work-stealing schedulers. In a naive work-stealing scheduler the idle processors can sometimes gang up on the busy processors, swamping them with requests for work, and preventing them from getting any work done. I solved this problem in the StarTech scheduler by using a globally synchronous throttling strategy (see Section 4.7.) The swamping problem can be understood using queueing theory.

We not only wish to achieve good time performance, but we must not use too much memory. For example, a breadth-first greedy search of the game tree is one of the schedules that achieves Brent's bound, but the amount of memory needed is nearly proportional to the amount of total work. Blumofe and Leiserson [BL93], inspired by memory exhaustion problems I had in early versions of StarTech, showed that for certain kinds of dataflow graphs, such as the dataflow graph for Jamboree search, there are easy-to-find schedules that not only achieve Brent's time bounds, but also use no more memory per processor than a serial schedule uses. A serial schedule of Jamboree search uses memory which is linear in the depth of the game-tree. All known serial game-tree search algorithms use at least that much memory. This chapter includes a study of the performance impact of the decisions made to guarantee that StarTech's space bounds are kept reasonable.

In summary, this chapter presents a performance study of the StarTech chess program.

- We start by studying the performance of the chess algorithms. Section 5.2 analyses the performance of Jamboree search on best-ordered uniform game trees. Section 5.3 analyses the performance Jamboree search on worst-ordered uniform game trees. Section 5.4 studies the performance of Jamboree search on real chess positions and discusses discusses some of the interactions between the algorithm and the scheduler that can make it difficult to understand the performance of Jamboree search.

- We then move on to study the StarTech scheduler. Section 5.5 shows why scheduling chess programs is demanding. Section 5.6 studies the performance of the scheduler relative to critical path and total work. Section 5.7 studies the swamping problem using queueing theory. Section 5.8 concludes the scheduler study by discussing the space-time tradeoffs implied by the decision I made to fix the processing of each position in the chess tree onto a single processor, rather than allowing a position to migrate after its creation from one processor to another.

In summary, to quantify the behavior of a parallel algorithm, we use three measures of parallel computation: the critical path length, the work, and the work efficiency. Critical path length and total work provide us with bounds on the performance of the computation, while work-efficiency provides us with a measure of how many processors we need to employ to overcome the overhead of running a parallel algorithm. By curve-fitting the performance of the system to the measured critical path and work, we can determine the overheads induced by communications, load-balancing, and scheduling. These measures are fundamental to a parallel computation and provide genuine insight into how a program behaves.

5.2 Analysis of Best-Ordered Trees

To analyze best-ordered trees, we borrow some notation from Knuth and Moore [KM75]. Knuth and Moore proved that for a best-ordered tree, α-β search searches exactly the *critical tree*. To define the critical tree, we first define three three position *types*: Type 1, Type 2, and Type 3.

- The root of the game tree tree is a Type 1 position.
- The first child of a Type 1 position is a Type 1 position. The other children of a Type 1 position are Type 2 positions.
- The first child of a Type 2 position is a Type 3 position. The other children of a Type 2 position are not given types.
- All the children of a Type 3 position are Type 2 positions.

The critical tree is the set of all positions of Type 1, Type 2, or Type 3. Figure 5-1 shows a uniform game tree of depth 3 in which each position has 3 children, with the critical tree highlighted. The types of the positions in the critical tree are shown. It turns out that Jamboree also searches exactly the critical tree in a best-ordered game tree, and it searches each position exactly once. Thus, if we analyze the critical tree, we have analyzed the entire tree.

Definition 5.1 *We denote the best-ordered uniform game-tree of height h with d children at each internal position as $\mathcal{B}_{d,h}$. The value of $\mathcal{B}_{d,h}$ must be finite.*[2]

Definition 5.2 *We denote the critical tree of $\mathcal{B}_{d,h}$ as $\mathcal{C}_{d,h}$.*

[2]Knuth and Moore note that if a game tree has a score that is $+\infty$ or $-\infty$, the combinatorics work out slightly differently.

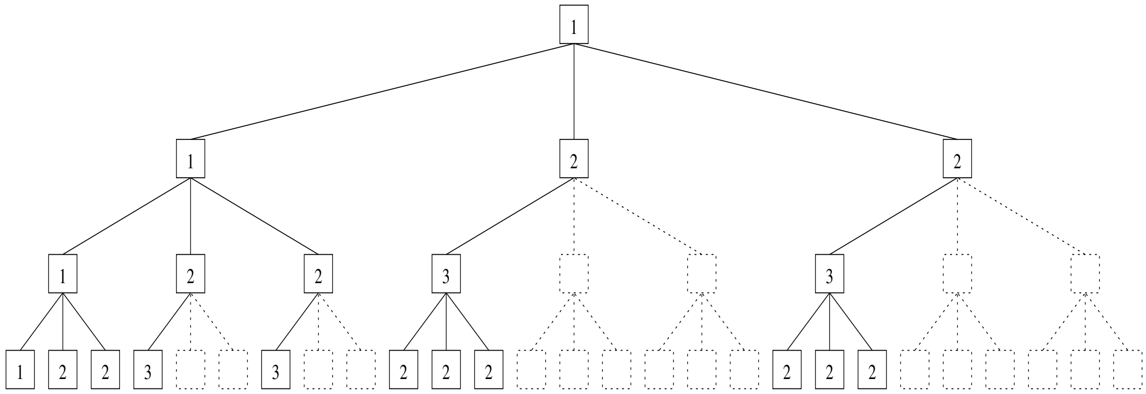


Figure 5-1: The critical tree of a best-ordered uniform game-tree. The critical tree is shown with solid lines, while the rest of the tree is shown with dotted lines. Also shown are the Knuth-Moore types of the positions that are in the critical tree.

For our analysis of uniform trees, we assume that each position costs one unit of time to visit. We also assume that all of the time is spent when the position is first visited. This lumped-cost assumption is close to the reality of StarTech, where move generation and static evaluation take much more time than the time taken between children. Almost the only thing to do between children is to take the maximum of the best known score with the score of a child. Also, we can charge any between-children costs to the children. Knuth and Moore analyzed the number of leaves in the critical tree, which is

$$\text{Leaves of } \mathcal{C}_{d,h} = d^{\lceil h/2 \rceil} + d^{\lfloor h/2 \rfloor} - 1.$$

We are interested in the number of internal positions of the tree.

Theorem 5.3 *For any $d \geq 2$ and any $h \geq 1$, we have*

$$|\mathcal{C}_{d,h}| = \begin{cases} \dfrac{3d+1}{d-1} d^{h/2} - \dfrac{d+3}{d-1} - h - 1 & \textit{if } h \textit{ is even,} \\[2ex] \dfrac{d^2+3d}{d-1} d^{(h-1)/2} - \dfrac{d+3}{d-1} - h - 1 & \textit{if } h \textit{ is odd.} \end{cases} \tag{5.5}$$

The proof follows in a few paragraphs, but first let us investigate the ramifications of Theorem 5.3. We can use Equation 5.5 to determine the work done by Jamboree search on a best-ordered uniform tree, as well as the critical path of the computation and the average available parallelism.

Theorem 5.4 *If $d > 1$, Jamboree search of $\mathcal{B}_{d,h}$*

- *performs work $|\mathcal{C}_{d,h}|$,*
- *has critical path length $|\mathcal{C}_{2,h}|$,*
- *has average available parallelism $|\mathcal{C}_{d,h}|/|\mathcal{C}_{2,h}|$.*

Proof of Theorem 5.4: Knuth and Moore showed that α-β search searches exactly $\mathcal{C}_{d,h}$. For best ordered trees, Jamboree searches the tree that α-β would search, since every test succeeds. The

h	$\lvert\mathcal{C}_{36,h}\rvert/\lvert\mathcal{C}_{2,h}\rvert$	$\lvert\mathcal{C}_{36,h}\rvert/\lvert\mathcal{C}_{3,h}\rvert$
0	1.0	1.0
1	12.3	9.2
2	18.0	12.0
3	130.8	72.0
4	223.9	108.9
5	1792.4	722.0
6	3302.1	1162.3
7	27933.8	8067.1
8	53375.4	13309.9
9	464666.1	94101.4
10	905331.4	156793.5

Figure 5-2: Numerical values for the average available parallelism for best-ordered uniform game tree of degree 36 for various heights. The first column is the height. The second column is the average available parallelism for the Jamboree algorithm valuing one child before testing the rest in parallel. The third column is the average available parallelism for the Jamboree algorithm serially valuing two children and then testing the rest in parallel.

critical path is the size of $\mathcal{C}_{2,h}$, because the first child is searched, then all the other children are searched in parallel. Thus, we can ignore all but the first and second child, and they are done serially. The average available parallelism is computed by definition. $\square$

For chess mid-game positions, it is estimated [Sha50] that the average number of children of a position is about 36. Plugging $d = 36$ into Equation 5.5 yields the values shown in Figure 5-2. We hope to search to depth 10 or deeper, and there is plenty of parallelism if we can make the search best-ordered. The average available parallelism for a version of Jamboree search that values the first two children serially before testing the rest of the children in parallel is also shown in Figure 5-2. Doing two children serially before testing the rest should reduce the amount of work done (since we are less like to do a lot of work in parallel that turns out to be unnecessary) and reduces the parallelism. If we can achieve good enough move ordering, there appears to be plenty of parallelism even for this more serial version of Jamboree search.

The number of children valued serially can be viewed as a tuning parameter. Charles E. Leiserson suggested a strategy in which the number of children valued serially is a function of the depth of the search. The idea is to find more parallelism for shallow searches while improving the work efficiency for deep searches. My experiments with this strategy were not conclusive, although we shall see, in Section 6.5, that a related idea works quite well.

Proof of Theorem 5.3: We partition $\mathcal{C}_{d,h}$ into disjoint subtrees $\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_h$ as shown in Figure 5-3 for the example $\mathcal{C}_{3,5}$. We let $V_i = |\mathcal{V}_i|$, and the recurrence relation for V_i is

$$V_i = \begin{cases} 1 & \text{if } i = 0, \\ d & \text{if } i = 1, \\ d \cdot V_{i-2} + d - 1 & \text{if } i > 1. \end{cases} \tag{5.6}$$

This recurrence is derived by observing that for $i > 1$, $\mathcal{V}_i$ can be assembled from d copies of $\mathcal{V}_{i-2}$ plus $d - 1$ vertices. To perform this assembly, observe that the root of $\mathcal{V}_i$ has $d(d-1)$ great-grandchildren that are each identical to each of the $(d-1)$ children of the root of of $\mathcal{V}_{i-2}$. Thus d copies of $\mathcal{V}_{i-2}$ accounts for all of the great grandchildren of $\mathcal{V}_i$, with d nodes left over. We

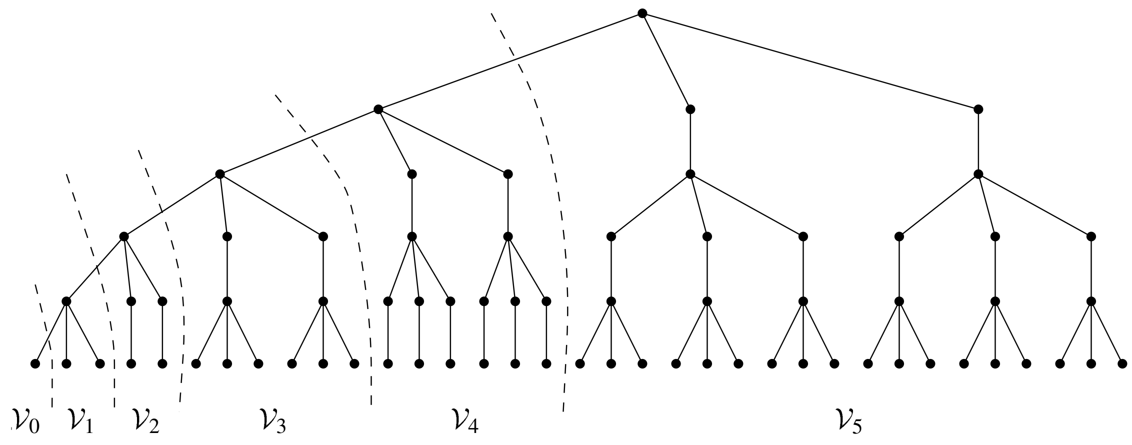


Figure 5-3: To solve for the number of vertices in the critical tree of a best-ordered uniform game tree, we partition the tree as shown, and then compute $\sum_{i=0}^{h} |\mathcal{V}_i|$.

have not accounted for the root of $\mathcal{V}_i$, its $d-1$ children, or its $d-1$ grandchildren. Thus we need $2d-1$ more nodes, and we already have d left over from the d copies of $\mathcal{V}_{i-2}$. We add in the $d-1$ children needed to account for all of Vhi.

Linear recurrence 5.6 can be solved using generating functions. We are not really interested in solving for V_i, but we need the generating function for V_i to solve for $|\mathcal{C}_{d,h}|$. Recall that the generating function generating function $v(x) = \sum_{i=0}^{\infty} V_i x^i$ is an infinite symbolic polynomial in which the coefficient of x^i is the value of V_i. Recall also that

$$\frac{d-1}{1-x} = \sum_{i=0}^{\infty} (d-1)x^i,$$

and that

$$dx^2 v(x) = \sum_{i=0}^{\infty} dV_i x^{i+2},$$

so that with the corrections for the boundary cases thrown in, we find that $v(x)$ satisfies

$$v(x) = \frac{d-1}{1-x} + dx^2 v(x) + x + 2 - d. \tag{5.7}$$

Solving for $v(x)$ gives

$$v(x) = \frac{d-1}{(1-x)(1-dx^2)} + \frac{2-d+x}{1-dx^2}. \tag{5.8}$$

Putting $v(x)$ into the canonical form,

$$v(x) = \frac{a}{1-x} + \frac{bx+c}{1-dx^2} \tag{5.9}$$

yields the generating function

$$v(x) = \frac{-1}{1-x} + \frac{(d+1)x-2}{1-dx^2}. \tag{5.10}$$

Now to solve for $|\mathcal{C}_{d,h}|$ we multiply by $1/(1-x)$, which Liu [Liu68, Example 2-7] calls 'the summing operator'. The generating function for $|\mathcal{C}_{d,h}|$ is

$$
\begin{aligned}
s(x) &= \sum_{i=0}^{\infty}\left(\sum_{j=0}^{i} V_j\right) x^i && (5.11)\\
&= \frac{v(x)}{1-x} && (5.12)\\
&= \frac{d-1}{(1-x)^2(1-dx^2)} + \frac{2-d+x}{(1-x)(1-dx^2)} && (5.13)\\
&= \frac{(3+d)/(1-d)}{1-x} + \frac{-1}{(1-x)^2}\\
&\quad + \frac{(-3d-d^2)/(1-d)x + (-3d-1)/(1-d)}{1-dx^2}, && (5.14)
\end{aligned}
$$

i.e., the size of $\mathcal{C}_{d,h}$ is the sum of the sizes of the partition, $|\mathcal{V}_0| + \cdots + |\mathcal{V}_h|$.

We now need to calculate the coefficient of x^h in Equation 5.14 to determine $|\mathcal{C}_{d,h}|$. For the first term the coefficient is just $(3+d)/(1-d)$. The second term applies the summing operator to $\sum_{i=0}^{\infty}(-x^i)$ yielding

$$
\begin{aligned}
\frac{\sum_{i=0}^{\infty}(-x^i)}{1-x} &= \sum_{i=0}^{\infty}\left(\sum_{j=0}^{i}(-1)\right) x^i\\
&= \sum_{i=0}^{\infty}(-i-1)x^i.
\end{aligned}
$$

and the coefficient of x^h in the summed polynomial is $(-h-1)$. The third term can be expanded by substituting $y = dx^2$ into

$$\frac{1}{1-y} = \sum_{i=0}^{\infty} y^i,$$

and multiplying by the top yielding

$$
\begin{aligned}
&\frac{(-3d-d^2)/(1-d)x + (-3d-1)/(1-d)}{1-dx^2} =\\
&\sum_{i=0}^{\infty}\left(\frac{-3d-1}{d-1}d^i x^{2i} + \frac{-3d-d^2}{1-d}d^i x^{2i+1}\right). \qquad (5.15)
\end{aligned}
$$

The coefficient of x^h in Equation 5.15 is computed as follows:

- for even h, the only possible match is $h = 2i$ giving $(-3d-1)d^{h/2}/(d-1)$ as the coefficient of x^h.
- for odd h, the only possible match is $h = 2i+1$ giving $(-3d-d^2)d^{(h-1)/2}/(1-d)$.

Summing up the coefficients contributed by each term of Equation 5.14 yields Equation 5.5, proving Theorem 5.3. $\square$

5.3 Analysis of Worst-Ordered Game Trees

Surprisingly, for worst-ordered uniform game trees, the speedup of Jamboree search over serial α-β search turns out to be just under 1. That is, Jamboree search is worse than serial α-β search. For comparison, parallelized negamax search achieves linear speedup on worst-ordered trees, and Fishburn's MWF algorithm achieves not-quite linear speedup on worst-ordered trees [Fis84].

Definition 5.5 *We denote the worst-ordered uniform game-tree of height h with d children at each internal position as $\mathcal{W}_{d,h}$. The value of $\mathcal{W}_{d,h}$ must be finite.*

Theorem 5.6 *For Jamboree search on $\mathcal{W}_{d,h}$:*

- *The critical path length is*

$$C(\mathcal{W}_{d,h}) = \begin{cases} \dfrac{d^2(1+d)}{(1-d)(2-d^2)}d^h + \dfrac{6+4d}{2-d^2}2^{h/2} - \dfrac{2}{1-d} & \textit{if } h \textit{ is even,} \\[2ex] \dfrac{d^2(1+d)}{(1-d)(2-d^2)}d^h + \dfrac{8+6d}{2-d^2}2^{(h-1)/2} - \dfrac{2}{1-d} & \textit{if } h \textit{ is odd.} \end{cases} \tag{5.16}$$

- *The work performed is*

$$W(\mathcal{W}_{d,h}) = \begin{cases} \dfrac{3d}{d-1}d^h - \dfrac{1+3d}{d-1}d^{h/2} + \dfrac{d}{d-1} & \textit{if } h \textit{ is even,} \\[2ex] \dfrac{3d}{d-1}d^h - \dfrac{3+d}{d-1}d^{(h+1)/2} + \dfrac{d}{d-1} & \textit{if } h \textit{ is odd.} \end{cases} \tag{5.17}$$

Corollary 5.7 *Asymptotically, as d and h go to infinity, for Jamboree search on $\mathcal{W}_{d,h}$,*

- *the work efficiency as compared to serial α-β search is $1/3$,*
- *the average available parallelism is 3, and*
- *the speedup is bounded above by 1.*

In fact, as Figure 5-4 shows, the asymptotes are approached quite quickly for game trees of degree 36.

Proof of Corollary 5.7: The work performed by serial α-β search on worst-ordered trees is exactly $W_s = d^h$. For any fixed h, we have

$$\lim_{d\to\infty} \frac{W_s}{\mathcal{W}_{d,h}} = \frac{1}{3}.$$

The work is $W_\infty = 3d^{h+1}/(d-1) - \Omega(d^{h/2})$, and the critical path is $C = d^{h+2}(1+d)/((1-d)(2-d^2)) + O(2^{h/2})$. Thus, the average available parallelism is

$$\begin{aligned} W_\infty/C &= \frac{3d^{h+1}/(d-1) - \Omega(d^{h/2})}{d^{h+2}(1+d)/((d-1)(d^2-2)) + O(d^{h/2})} \\ &\le \frac{3}{d(1+d)/(d^2-2)} \\ &= \frac{3(d^2-2)}{d^2+d} \\ &\le 3. \end{aligned}$$

h	Work efficiency	Average available Parallelism	Speedup over Serial
0	1.000000	1.000000	1.000000
1	0.513889	1.894737	0.973684
2	0.342850	2.833819	0.971574
3	0.336708	2.885218	0.971477
4	0.333593	2.912146	0.971472
5	0.333426	2.913602	0.971471
6	0.333341	2.914351	0.971471
7	0.333336	2.914392	0.971471
8	0.333334	2.914413	0.971471
9	0.333333	2.914414	0.971471
10	0.333333	2.914414	0.971471

Figure 5-4: The work-efficiency, average available parallelism, and speedup over the serial code of Jamboree search on $\mathcal{W}_{36,h}$, the worst-ordered uniform game-tree in which each internal position has 36 children.

The time to execute the parallel code is bounded below by $C = d^h + O(d^h - 1)$, so the speedup, which is limited by W_s/C, approaches 1 from below. $\square$

Proof of Theorem 5.6: The critical path, C_h for $\mathcal{W}_{d,h}$ satisfies

$$C_h = \begin{cases} 1 & \text{if } h = 0, \\ 1 + dC_{h-1} + O_{h-1} & \text{if } h > 0; \end{cases} \tag{5.18}$$

where

$$O_h = \begin{cases} 1 & \text{if } h = 0, \\ 3 & \text{if } h = 1, \\ 3 + 2 \cdot O_{h-2} & \text{if } h > 1. \end{cases} \tag{5.19}$$

For the case of $h > 0$, this Equation 5.18 can be derived as follows:

- The 1 comes from searching the root of $\mathcal{W}_{d,h}$.
- Jamboree search, on a worst-ordered tree, values the first child, contributing C_{h-1}.
- Then, Jamboree search tests all the other children in parallel. Since the tree is uniform, the critical path of all the tests is the same, so just consider a single one of the tests. The critical path of a test that fails is given in Equation 5.19. For a test that fails, the first child is searched with a test that succeeds, then all the other children are searched with a test that succeeds.
- Finally after all the tests fail, the rest of the children are serially researched contributing another $(d - 1)C_{h-1}$.

To solve Equation 5.18 we again use generating functions. The generating function for O_h is

$$O(x) = \frac{3}{1 - x} + 2x^2 O(x) - 2, \tag{5.20}$$

which yields

$$O(x) = \frac{1+2x}{(1-x)(1-2x^2)}. \tag{5.21}$$

The generating function for C_h is

$$C(x) = \frac{1}{1-x} + dxC(x) + xO(x), \tag{5.22}$$

which yields

$$\begin{aligned} C(x) &= \frac{1-x}{(1-dx)(1-x)(1-2x^2)} && (5.23) \\ &= \frac{d^2(1+d)/((1-d)(2-d^2))}{1-dx} + \frac{-2/(1-d)}{1-x} \\ &\quad + \frac{(6+4d)/(2-d^2) + (8+6d)x/(2-d^2)}{1-2x^2}. && (5.24) \end{aligned}$$

Reading the coefficients from Equation 5.24 as we did on Page 80, gives

$$C_h = \begin{cases} \dfrac{-2}{1-d} + \dfrac{d^2(1+d)}{(1-d)(2-d^2)}d^h + \dfrac{6+4d}{2-d^2}2^{h/2} & \text{if } h \text{ is even,} \\ \\ \dfrac{-2}{1-d} + \dfrac{d^2(1+d)}{(1-d)(2-d^2)}d^h + \dfrac{8+6d}{2-d^2}2^{(h-1)/2} & \text{if } h \text{ is odd.} \end{cases} \tag{5.25}$$

To get the formula for the work done on worst-ordered trees, we use this linear recurrence for W_h, the work done on a tree of depth h:

$$W_h = \begin{cases} 1 & \text{if } h = 0, \\ 1 + dW_{h-1} + (d-1)A_{h-1} & \text{if } h > 0; \end{cases} \tag{5.26}$$

where

$$A_h = \begin{cases} 1 & \text{if } h = 0, \\ d+1 & \text{if } h = 1, \\ d+1+dA_{h-2} & \text{if } h > 1. \end{cases} \tag{5.27}$$

We obtain these recurrence relations by observing that the Jamboree algorithm first searches the root of the tree (1), then searches the first child (W_{h-1}), then tests each of the other children (dA_{h-1}), then researches each of the tested children $(d-1)W_{h-1}$. The amount of work done when testing a tree of depth $h > 1$, when the test fails, is 1 for the root, d for each of the children each of has the test succeed, so they only test one grandchild each, and each grandchild's test fails (dA_{h-2}.)

The generating function for A_h is

$$\begin{aligned} A(x) &= \frac{d+1}{1-x} + dx^2A(x) - d && (5.28) \\ &= \frac{1+dx}{(1-x)(1-dx^2)} && (5.29) \end{aligned}$$

The generating function for W_h is

$$W(x) = \frac{1}{1-x} + dxW(x) + (d-1)xA(x) \tag{5.30}$$

$$= \frac{1 - x + dx + 2dx^2 + d^2x^2}{(1-x)(1-dx)(1-dx^2)} \tag{5.31}$$

$$= \frac{d}{d-1} \cdot \frac{1}{1-x} + \frac{3d}{d-1} \cdot \frac{1}{1-dx} + \frac{d(3+d)x + (1+3d)}{1-d} \cdot \frac{1}{1-dx^2} \tag{5.32}$$

Reading the coefficients for $W(x)$ (as we did on Page 80) gives

$$W_h = \begin{cases} \dfrac{d}{d-1} + \dfrac{3d}{d-1}d^i + \dfrac{1+3d}{1-d}d^{i/2} & \text{if } h \text{ is even,} \\ \\ \dfrac{d}{d-1} + \dfrac{3d}{d-1}d^i + \dfrac{d(3+d)}{1-d}d^{(i-1)/2} & \text{if } h \text{ is odd;} \end{cases} \tag{5.33}$$

thus proving the theorem. □

5.4 Jamboree Search on Real Chess Positions

We now understand the Jamboree search for two special cases of search trees: best-ordered and worst-ordered uniform trees. Real chess trees are more complex, however. They have nonuniform depth and degree, and they are 'mostly' best ordered. In this section we study the critical path length, total work, and work efficiency for real chess trees. We start with a discussion of the difficulties with measuring Jamboree search on real chess trees. We then show that for real chess trees the critical path is short compared to the total work, which means that there is plenty of parallelism and that the total work produced by the algorithm compares favorably to a serial implementation.

It is difficult to analyze Jamboree search for arbitrary game trees, because it is difficult to characterize the tree itself, and the tree that is actually searched can depend on how the work is scheduled. Unlike many other applications, the shape of the tree traversed by Jamboree search can be affected by the order of the execution of the work, sometimes increasing the work and sometimes decreasing work. Thus, measurements of "critical path length" and "work" on a particular run may be different than the measurements taken on another run, because the trees themselves are different. It is not clear what "critical path" and "work" mean for Jamboree search on arbitrary trees.

The work, as measured on one run, might be more or less than the work on another run. Sometimes more work is done by a fully parallel Jamboree search than by serial scheduling of Jamboree search. For example, as shown in Figure 5-5, if the second subtree of a position fails high, fully-parallel Jamboree search may still search the third and subsequent subtrees, whereas a serial scheduling would allow Jamboree to stop after examining the second child. More generally, Jamboree search might start searching the later subtrees and then abort those searches partway through. A serial schedule would never have started searching those later subtrees.

Sometimes less work is done by fully parallel Jamboree search than would be done under a serial schedule. Figure 5-6 shows such an example. In this case, the serial schedule finishes searching the large subtree E before quickly finding out that subtree F would fail high. Under a fully parallel schedule, on the other hand, the scheduler might expand the search of D and F in parallel, obtain the result for F, and abort the subsearch of E. If the tree had been best-ordered, both the serial and the parallel algorithm would be guaranteed to do the same amount of work. In this case,

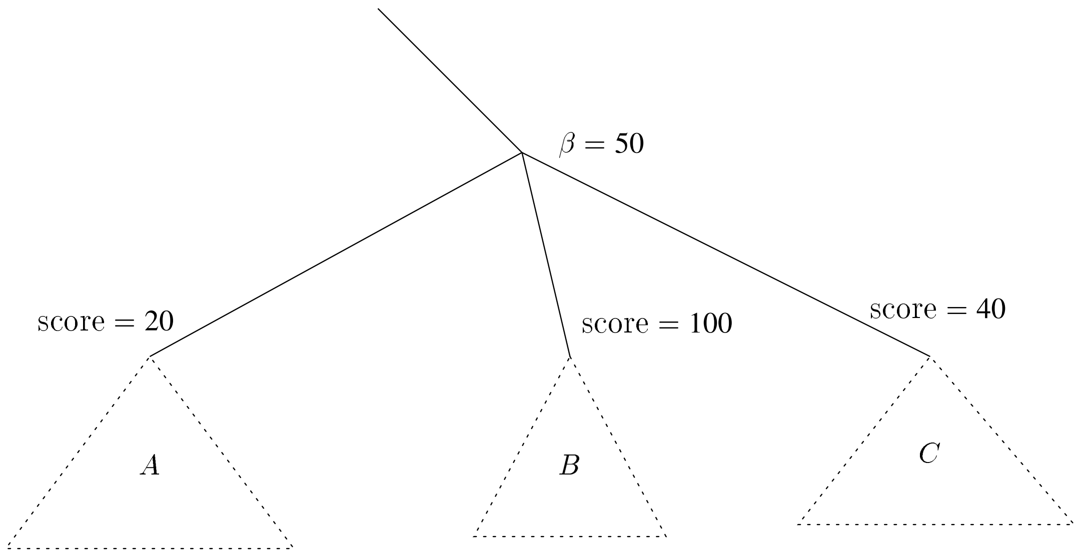


Figure 5-5: Jamboree search sometimes does more work than a serial algorithm. In this case the value of β is 50, so as soon as a child is found with score $\geq$ 50, a serial algorithm would stop examining children. Jamboree search, like a serial algorithm, searches the first child, A, obtaining a value of 20. Then Jamboree searches B and C in parallel, aborting the search of C only when the value of B is determined. A serial algorithm would never search C.

the breadth-first search order used by a fully parallel schedule is buying performance. The work done by one scheduling of Jamboree search is generally incomparable to the work done by another scheduling.

The situation for the critical path length is a little better than for the estimate of work. Measuring the "critical path length" using any schedule we want yields an upper bound to the true infinite-processor critical path length.

The critical path length of a computation is generally accepted to be the time it takes for an infinite number of processors with a perfect scheduler to execute the program. I obtain an estimate to the infinite-processor critical-path length while running the StarTech program by using a time-stamping scheme. We can view Jamboree search as a dataflow graph, as shown in Figure 5-7. The arcs of the dataflow graph carry control information, so that when a token arrives at a procedure call, the procedure may start, and when the procedure finishes, a token is delivered at the output of the call. In general, one can determine the time that any particular instruction would execute on perfect infinite-processor schedule by time-stamping each token. As shown in Figure 5-8, the time-stamp of a token on the output of an instruction can be computed by taking the maximum of the time-stamps of the tokens on the inputs of the instruction and adding the time it takes to execute the instruction.

My measurements indicate that 85% to 95% of the positions encountered in the tree by StarTech are best-ordered. That is, the move-ordering heuristics of StarTech guess which move is the best move about 85% to 95% of the time. The precise rate depends on which position being searched.

For Jamboree search, the measured critical path length is an overestimate of the critical path length for an infinite processor machine with a perfect greedy scheduler. To see why this is so, consider the dataflow graph shown in Figure 5-9, which includes a `first` operator that passes the first result that arrives to its output. When one subgraph completes, the other subgraph can be

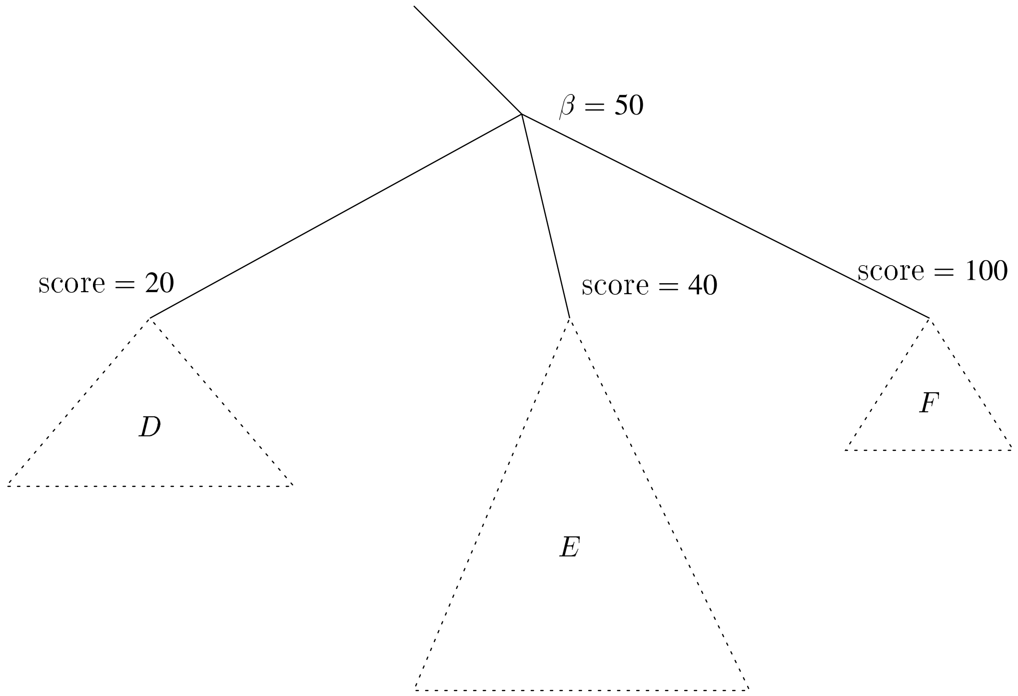


Figure 5-6: Jamboree search sometimes does less work than a serial algorithm. The value of β is 50. The first child, D is searched and the score is determined to be 20. The second child, E, is a large subtree with a score of 40. Jamboree search might search the third child, F, which has a score of at least 100, before E has been finished, and Jamboree might abort the expensive search of E.

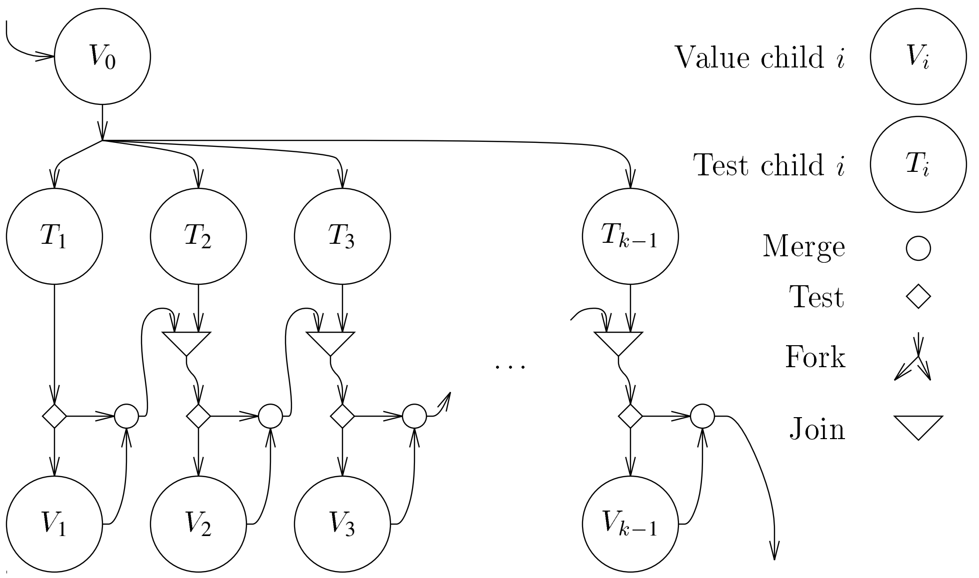


Figure 5-7: The dataflow graph for Jamboree search. First Child 0 is searched to determine its value, then the rest of the children are tested in parallel to try to prove that they are worse choices than Child 0, and then each of the children that fail their respective tests are serially researched. This dataflow graph can be used to measure the critical path length of the computation by using time-stamping. Compare this description of the Jamboree algorithm to the textual description in Figure 4-5.

aborted, since the `first` operator ignores its result. The infinite processor critical path of this dataflow graph is the minimum of the critical paths of the two subgraphs. An implementation that aborts the second subcomputation may be throwing away the result with the shorter critical path, and then the only critical-path length information it has will be an overestimate. Such a situation arises in Jamboree search after the search of the first child is complete and the rest of the children are searched in parallel. Some of those children may fail high, aborting all of the rest of the children, and the only critical path information available is an overestimate of the critical path. Thus, the empirical critical path length measurements overestimate the run time of the program on an infinite processor machine with a perfect greedy scheduler. Fortunately, my empirical measurements turn out to be good enough to allow us to tune the program.

I measured the performance[3] of StarTech running on a problem set of 25 chess positions designed by International Master L. Kaufman. Several other chess problem sets have appeared in the literature to test the skill of a chess program. The Bratko-Kopec set [KB82], one of the earliest test sets published for computers, was designed to show the deficiencies of a program rather than to estimate the program's strength. Feldmann *et al.* [FMM90] found that the Bratko-Kopec test set could not differentiate between master-level chess programs, and so they picked a collection of positions from actual games they had played to measure the performance of their program. Kaufman's problem set [Kau92, Kau93] was specifically designed to estimate the rating of

[3]I measured the run-time, the total work and, the critical path length, using the fast user-level timers described in Section 3.3. To avoid the problems described in Section 3.3 I performed the experiments on the CM-5 in dedicated (single-user) mode. To measure the critical path length, I used the timestamping technique described above.

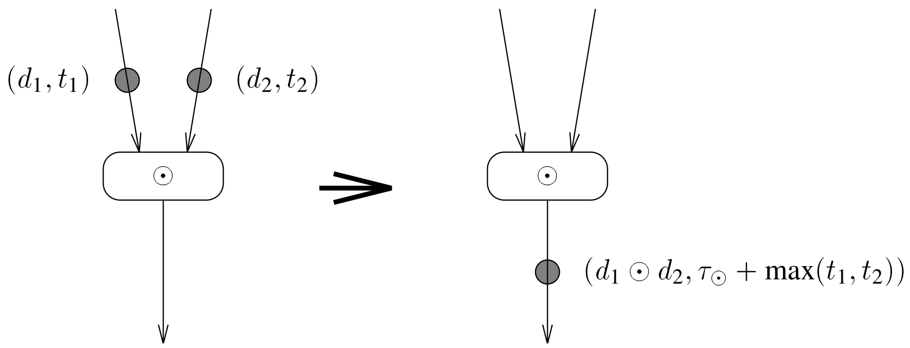


Figure 5-8: The time at which an instruction in a dataflow graph is executed in a perfect infinite-processor schedule can be computed by timestamping the tokens. In addition to the normal data-value of a token (d_1, d_2, and $d_1 \odot d_2$ respectively in the figure), the token includes a timestamp (t_1, t_2, and $\tau_\odot + \max(t_1, t_2)$ respectively.) The timestamp on the outgoing token is computed as a function of the timestamps of the incoming tokens and the time to execute the instruction.

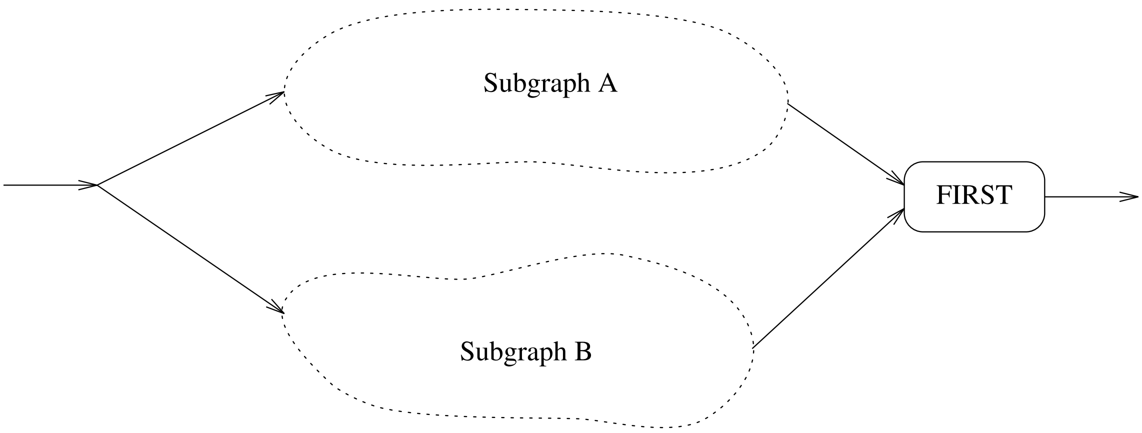


Figure 5-9: Two subgraphs of a dataflow graph compute in parallel. One keeps only the first result, aborting the computation in progress on the second subgraph. In this situation a reasonable estimate of the critical path is the critical path of the completed subgraph, even though that may be an overestimate. For example it is possible that Subgraph A finishes its computation first even though the critical path of Subgraph B is shorter than of Subgraph A.

a program that plays master-level chess. Kaufman uses 25 chess positions (20 tactical, 5 positional), each of which has a 'correct' answer. The program is timed on each position to determine how long it takes to decide to play correct move.[4]

The measured work increases with the size of the CM-5 on which we the program is running. Figure 5-10 shows how the measured work varies with the machine size for each of several executions of each of the 25 chess positions. The work efficiency of the program running on large machines does better on longer runs than on shorter runs. Figure 5-12 shows a scatter plot of the work

[4]For a precise explanation of Kaufman's test, see [Kau92, Kau93].

efficiency against the serial run time, for each of the machine sizes. For the larger machines, the work efficiency never is as good as for the smaller machines, but the program does always run faster on a big machines. The sample correlation coefficient (see for example [HL93, page 51])

$$r = \frac{\sum x_i y_i - (\sum x_i)(\sum y_i)/n}{\sqrt{\left[\sum x_i^2 - (\sum x_i)^2/n\right]\left[\sum y_i^2 - (\sum y_i)^2/n\right]}}$$

of the work efficiency to the run time is shown for each machine size. Thus, while the efficiency drops as the machine gets larger, if we fix the machine size, and let problems run longer, the efficiency improves.

Surprisingly, the measured critical path length increases as the number of processors grows. Figure 5-11 shows how the measured critical path length varies with the machine size for each of 25 different chess positions. We argued above that the critical path length is a lower bound to the infinite processor critical path length.

I believe that the increase in measured work and measured critical path length can be largely explained by transposition table effects. StarTech uses a global transposition table, the implementation of which is outlined in Section 6.3, to cache the results of recent searches. The transposition table improves efficiency in two ways. The first efficiency improvement is due to the reconvergence of the search. The same position may be encountered along two different variations, in which case the transposition table can immediately return the result for the second search without performing the search. The second efficiency improvement is due to the use of the transposition table as a move ordering heuristic. A previous search of the same position to a shallower depth is not good enough to shortcut the search, but StarTech can use the information about which move was best for the shallower search to improve the probability that StarTech considers the best move on the deeper search. In a larger machine, the transposition table is less effective than in a smaller, since a search that is performed serially on a smaller machine may be performed in parallel on a larger machine. On the other hand, the larger machine has a larger transposition table, which lowers the probability of some useful value being removed due to hash-key collisions.[5]

These effects, plus the fact that my critical path measurements do not accurately reflect dependencies through the transposition table, mean that for larger machines, the measured critical path tends to increase. In contrast, we argued earlier that the measured critical path length is an upper bound to the infinite processor critical path. The global transposition table is essential to the performance of the program, but the nondeterminacy introduced by the table can make it difficult to understand the program.

The scheduler and the Jamboree algorithm have positive interactions. Abstractly, the scheduler and the algorithm are separated. But in practice, there are interactions between them. If there is more parallelism than there are processors, then processors tend to do their work locally, effectively creating a larger grain size, and the efficiency of the underlying serial algorithm becomes the determining performance factor. The StarTech scheduler attempts to steals work that is near the root of the game tree, rather than work that is near the leaves. By stealing work near the root of the game tree, the size of stolen work is increased. On the other hand, if work is stolen that later is determined not to have been useful, more processor cycles are wasted. I found that for a given tree search, the average size of stolen work is larger for smaller machines.

In summary, the the critical path length and total work are stable enough to be useful for algorithm

[5]To make the transposition table equally effective for large and small machines, we shall explore, in Chapter 6, a transposition table modification called *deferred-reads*.

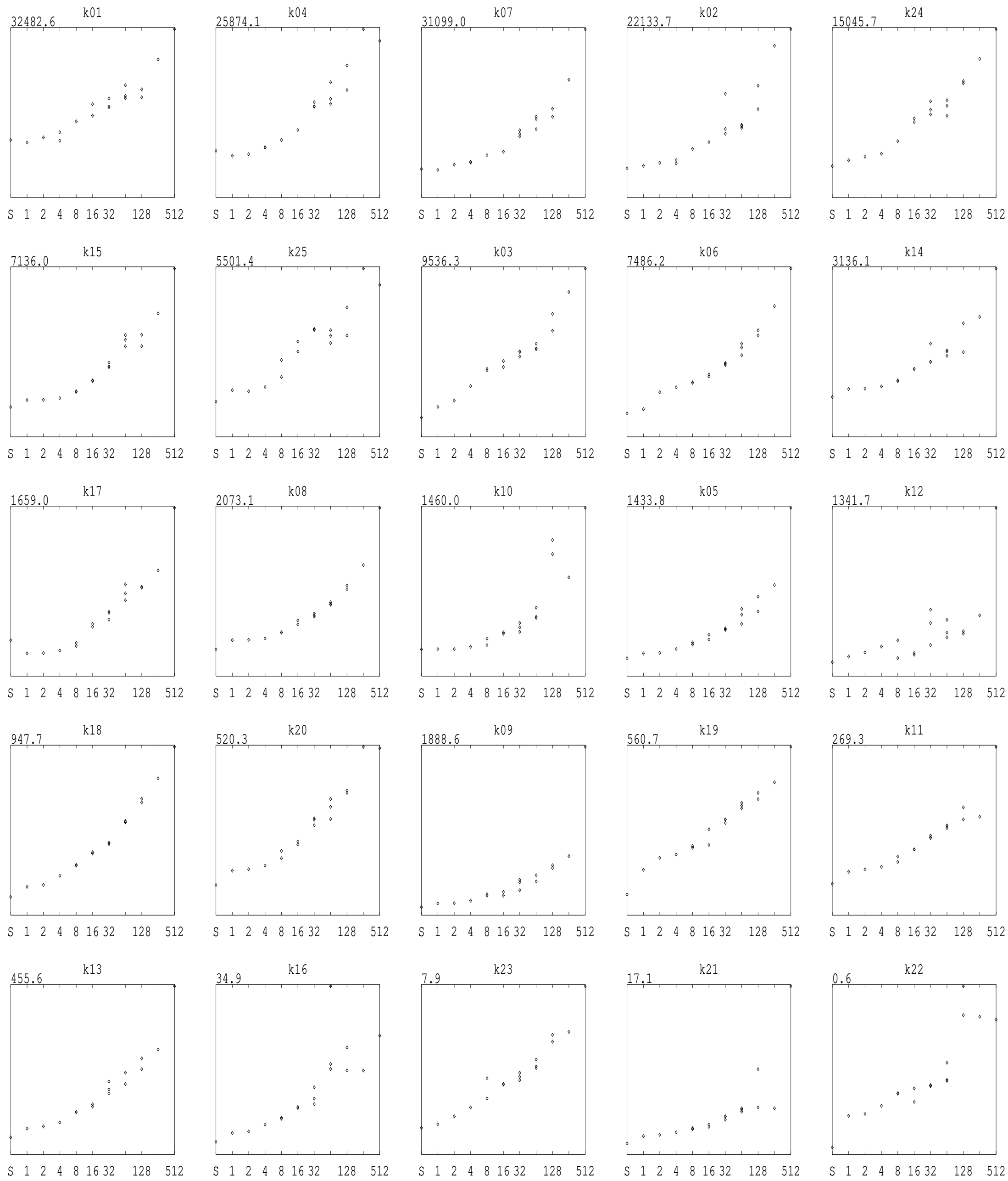


Figure 5-10: The total work of each of Kaufman's 25 test positions, as measured on various machine sizes. Each box represents one test position. The positions are named `k01` through `k25`. The horizontal axis on each graph is the machine size ('S' denotes my best serial implementation). The vertical axis is the total work executed, in processor-seconds. The range of the total work for each position is shown at the left, just above the graph for that position. The vertical axis is scaled to that range. Each plotted point corresponds to a single measured execution. The positions are plotted in descending order according to the time taken by the serial implementation.

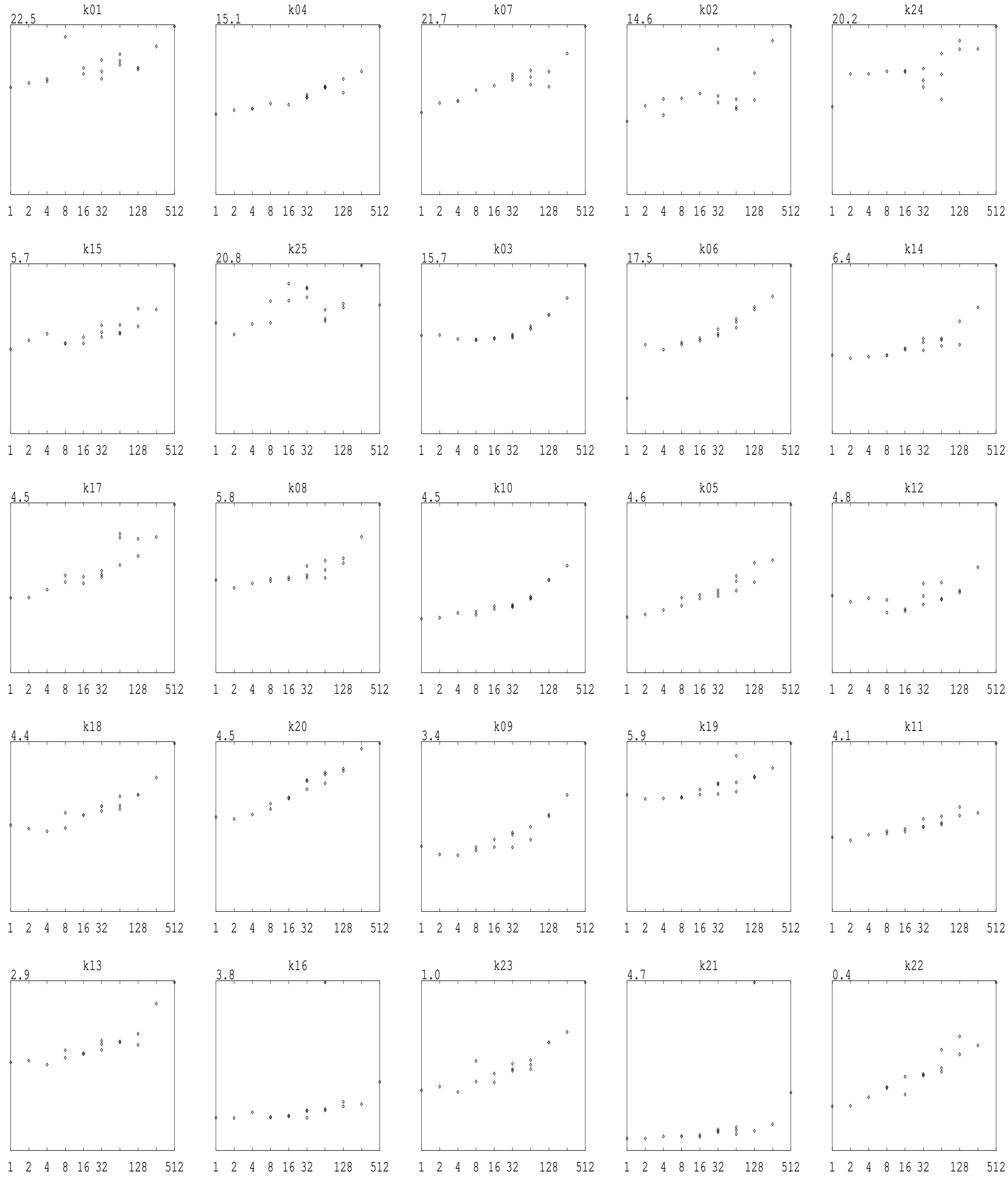


Figure 5-11: The critical path of each of Kaufman's 25 test positions, as measured on various machine sizes. Each box represents one test position. The positions are named `k01` through `k25`. The horizontal axis on each graph is the machine size. The vertical axis is the critical path length, in seconds. The range of the critical path length for each position is shown at the left, just above the graph for that position. The vertical axis is scaled to that range. Each plotted point corresponds to a single measured execution. The positions are plotted in descending order according to the time taken by the serial implementation.

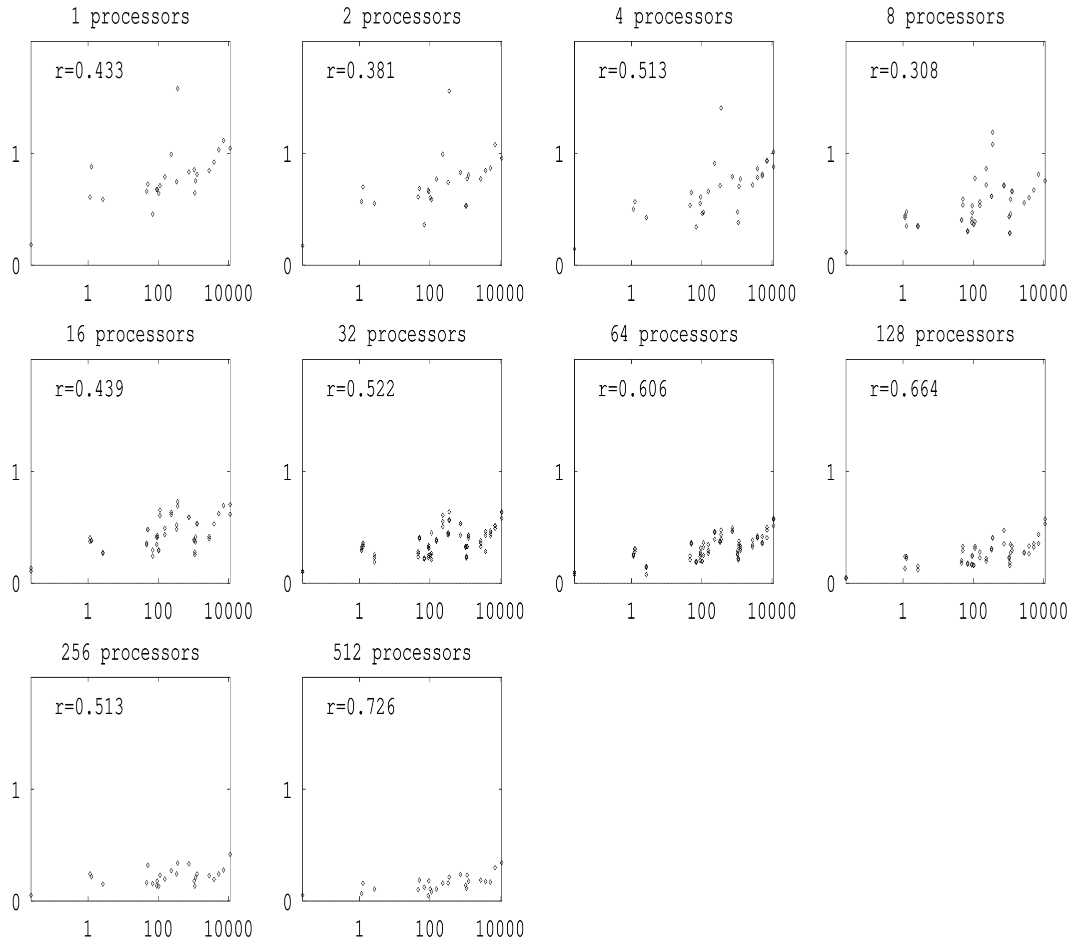


Figure 5-12: A scatter plot of the serial run time against the work efficiency. The horizontal axis of each scatter plot is the serial run time in seconds. The vertical axis is the work efficiency (the ratio of the serial run time to the parallel work.) The scales of both axes are the same for each scatter plot. One scatter plot is shown for each machine size. The sample correlation coefficient (r) is shown in the upper left hand corner of each scatter plot.

design, even though they do vary with the machine size. The average available parallelism and work efficiency of StarTech are both good enough to achieve significant speedup on chess problems.

5.5 Scheduling Parallel Computer Chess is Demanding

At this point, we have found that the performance of StarTech, as measured by critical path and total work, is quite promising. Before studying the performance of the scheduler, this section presents evidence that scheduling parallel computer chess is potentially very difficult. Other researchers have studied the performance of schedulers on easy problems. For example, using a highly parallel, uniform, program such as `nfib`[6] to measure the success of a scheduler does not really present compelling evidence that the scheduler actually addresses any difficult scheduling problems. Our evidence that computer chess is difficult to schedule comes from an examination of the ideal-parallelism profile of StarTech.

Consider the ideal-parallelism profile, shown in Figure 5-13. On a parallel machine with an unbounded number of processors and no scheduling overhead, the program would take 39 seconds to run. The total work (which is the area under the curve) is about 10000 processor-seconds. The total work is comparable to the time taken by the serial program, which is about 9000 seconds using my best serial implementation running on a single processor of the CM-5. While the average available parallelism is $10000/39 = 256$, for most of those 39 seconds, there is very little available parallelism. Figure 5-14 shows the data-points of the parallelism profile sorted from least-parallel to most-parallel, and plotted on a logarithmic scale, to make it easier to see how much of the 39 seconds is spent with how much parallelism. For a total of 18.4 seconds, the available parallelism is less than 16, and for 8.5 seconds the parallelism is less than 4. It is important that the program do a good job during those serial sections so that not much real wall-time is used up on the serial parts of the computation.

In general, an ideal-parallelism profile shows how a program would run on an infinite number of processors with perfect load balancing. The only constraint on the time-to-completion is the depth of the data-dependency graph. The horizontal axis is the time on the ideal machine. The parallelism curve shows how much parallelism is available at any given time, i.e., how many processors can be used at any given time. The length of the horizontal axis (the *critical path length*) shows how long the program runs (39 seconds in this case). The area under the curve (about 10000 processor-seconds) is the total number of processor-seconds that processors actually do useful work. The average available parallelism is the total work divided by the critical path length (in this case, 256 processors). As we shall see, for this example one may reasonably hope to use up to 256 processors at 50% efficiency, i.e., to finish the actual execution in $39 \cdot 2 = 78$ seconds.

The concept of an "ideal-parallelism profile" does not quite apply directly to Startech, since the work that is actually done depends on the scheduler. The program produces the same answer every time, but it may get that answer by expanding different trees, the details of which depend on low-level timing effects, on the number of processors, and on randomization in the search algorithm itself. One can really only compute an approximation to the ideal-parallelism profile, but I have found that this approximation tends to be reasonably accurate as the machine size is varied and over multiple runs of the same program.

The scheduling of the ideal work onto processors is shown by the processor utilization profile in Figure 5-15. The sharp spikes in the parallelism profile translate into wide horizontal regions where the machine is working efficiently (this is the *saturated-processor* region of the processor

[6] The `nfib` program is a doubly recursive routine to compute Fibonacci numbers.

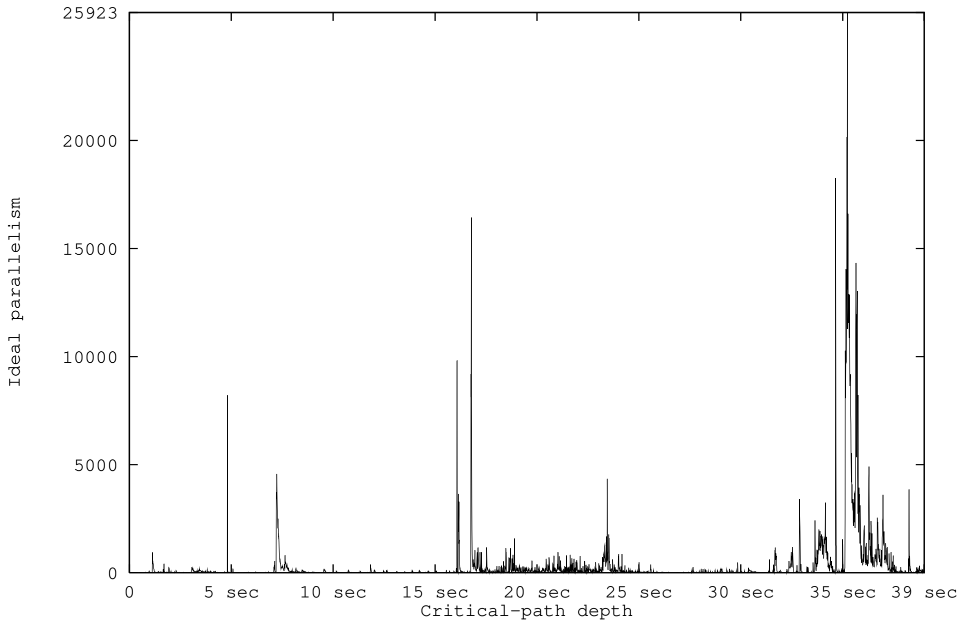


Figure 5-13: The ideal-parallelism profile of a typical chess position has many nearly serial sections. The ideal-parallelism profile is the number of threads that can be executed at any given moment in an ideal parallel machine with an infinite number of processors. The critical path length for this example is about 39 seconds, and the total work performed (the area under the curve) is about 10000 seconds.

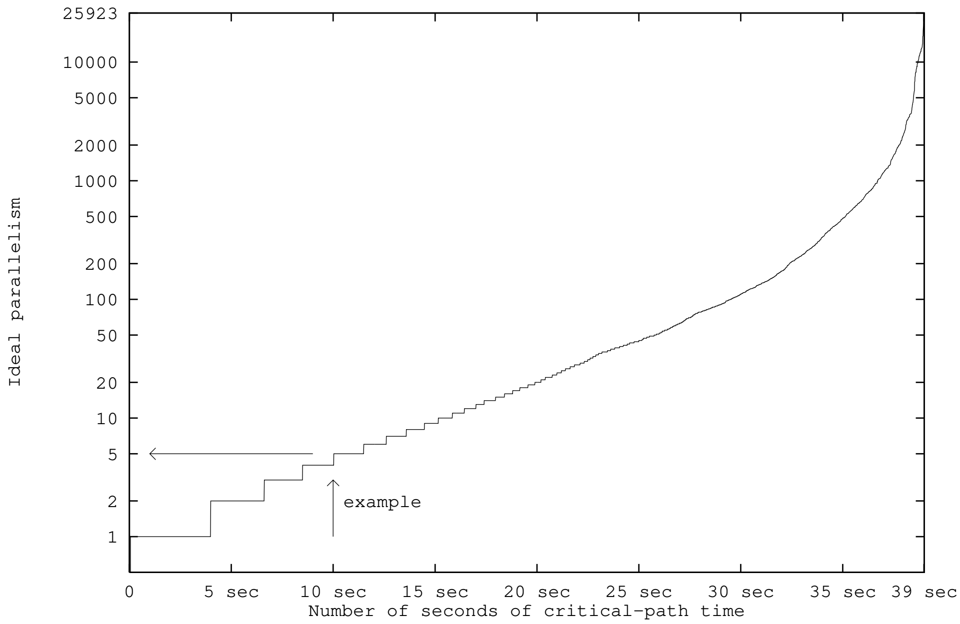


Figure 5-14: The sorted ideal-parallelism profile shows how the ideal parallelism is distributed. For example, (shown with arrows) for 10 seconds there is less than five-fold parallelism.

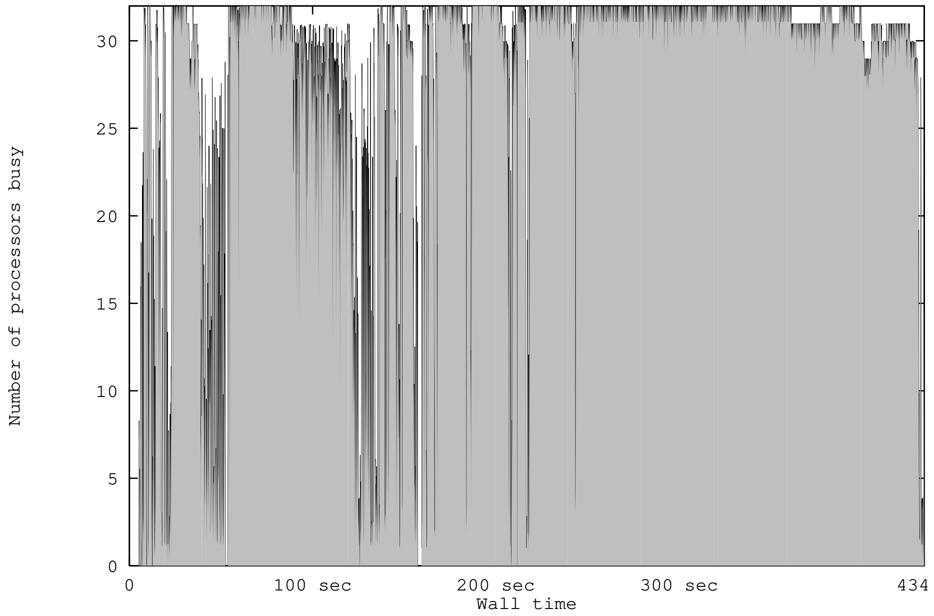


Figure 5-15: The processor utilization profile of StarTech running on a typical chess position. The horizontal axis is the wall time, and the height of the curve is the number of busy processors. The shaded area under the curve represents processor-cycles spent on useful work.

utilization profile). Note that even during the saturated processor region, there is a high frequency jitter resulting in something under a 10% inefficiency. In Section 5.8 we shall explore where this inefficiency arises. Note also that that the unsaturated regions of the processor utilization profile (which correspond to the regions of low parallelism in the ideal-parallelism profile) comprise a significant fraction of the work. The scheduler most do a good job during the unsaturated region.

The most natural way to express a parallel search is as an asynchronous program in which one logical process is associated with each position in the game tree and processes communicate with each other. The tree is too large to keep in memory, and so the tree must be searched with a constraint on how many positions can be expanded at any given time. While processing a position, operations must be performed on global data structures.

Thus, the issues for implementing a parallel computer chess program include the following. The game tree to be searched is nonuniform and too large to fit in memory. The best serial algorithms for game tree search are difficult to parallelize without losing their efficiency. The most effective parallel game-tree search algorithms are nondeterministic in that the precise tree searched depends on the number of processors, the message passing time, and many other factors. The parallel algorithms that seem to work well are sequential for large fractions of their critical paths. There are many factors in such a mishmash of calculation and communication that contribute or detract from the performance of the program.

The challenge is to take such a program and make it have predictable, high performance with small memory requirements. If the program runs slowly, we want to be able to understand why, so that we can improve it. My approach is to instrument the program to measure the total work and to estimate the critical path length. The critical path length and the total work give us bounds on the time to run the program on a given size machine. Although those measures are not precisely repeatable (because the tree itself may vary from run to run), they are consistent enough to give us a handle on the behavior of the system. Having characterized the algorithm, I address the scheduling issue by providing a scheduler, separate from the algorithm, which is guaranteed to use no more memory per processor than a serial implementation uses. In other words, we can make the chess program predictable by providing some measurement tools and some scheduling guarantees.

5.6 Performance of the StarTech Scheduler

Now that we have seen that computer chess has a demanding parallelism profile, we return to the simple summarization of the parallelism profile provided by the total work W and the critical path C. In this section we demonstrate that W and C can be used to predict the runtime of StarTech. As a consequence, for example, we can measure the program on a small machine and predict the performance on a big machine.

I measured the performance of StarTech running on P processors running on each of Kaufman's 25 test positions on each power-of-2 machine size from 1 to 512. I then performed a curve fitting using a weighted linear regression program[7] which minimized the least square difference

$$\sum_i \left(\frac{1}{\sigma_i}(\hat{T}_i - T_i)\right)^2,$$

where $\hat{T}_i$ is the value of T_i predicted by the model for the ith trial and $1/\sigma_i$ is a weight that allows me to minimize the relative errors. For example, I consider a prediction of 1100 seconds on a run of 1000 seconds to be as good as a prediction of 11 seconds on an actual run of 10 seconds. In both cases, the relative error is 10%. I set $\sigma_i = T_i$ to achieve this relative fit.

I found that the performance of StarTech is modeled by

$$\hat{T} = (4.303 \pm 0.167) + (1.024 \pm 0.019)\frac{W}{P} + (1.521 \pm 0.106)C \tag{5.34}$$

with a 95% confidence level. The model has the following statistical properties on the training data:

$$\begin{aligned} R &= 0.998717, \\ \text{MRE} &= 0.03855, \\ \text{MAXE} &= 0.6880, \end{aligned}$$

where R is the sample correlation coefficient, MRE is the mean relative error (the geometric mean of the relative errors), and MAXE is the maximum relative error. The maximum relative error tells us how bad the model is for the worst data point.

To test the performance model, I ran the StarTech program on a different collection of chess problems[8] on various machine sizes. The model provides an excellent match to the test data. The statistical properties for the model on the test data are

$$R = 0.999195,$$

[7] The weighted linear regression program was adapted by Eric A. Brewer from the singular value decomposition curve fitting code in *Numerical Recipes* [PTV*92].

[8] In this case, a collection of 20 problems was provided by H. Berliner.

$$\begin{aligned} \text{MRE} &= 0.061, \\ \text{MAXE} &= 0.880. \end{aligned}$$

I also tried improving the model by including more basis functions. I tried fitting the curve to the form

$$\begin{aligned} \hat{T} &= A_0 + A_1 W/P + A_2 C + A_3 P + A_4 C \lg P \\ &\quad + A_5 \max\{C, W/P\} + A_6 \min\{C, W/P\} + A_7 \cdot D + A_8 \cdot W \\ &\quad + \sum_{i=1}^{25} A_{(8+i)} k_i, \end{aligned} \tag{5.35}$$

where

- The A_i's are coefficients to be found via curve-fitting,
- $\hat{T}$ is the run time predicted by the model.
- W is the measured total work,
- C is the measured critical path length,
- P is the number of processors,
- D is the search-depth for a position, and
- k_i is a indicator variable for position i, i.e., $k_i = 1$ for tests on position i, and $k_i = 0$ for tests on position $j \neq i$.

For each of those terms, we can make an argument that they might matter. It might be that depending on the chess position, there is a different constant startup overhead. R. Blumofe [Blu94] made a theoretical argument that the critical path might need to be multiplied by $\lg P$. The lower bound arguments suggest that the minimum or the maximum of C and W/P might be important. The addition of all those produced a model with a sample correlation coefficient of 0.999168, a mean relative error of 0.02493, and a maximum relative error of 0.6749, all of which are only marginally better than the simpler model of Equation 5.34.

In order to demonstrate that none of those variables is really important, we can look at the residuals as function of each of those variables. The residual is the difference between the actual running time and the running time predicted by the model. The residuals should look like a random function of any particular variable. Figure 5-16 shows the relative residuals—the residual divided by the actual running time—plotted against all of the variables considered in the grandiose model of Equation 5.35. All of the position indicator variables k_i are shown on one graph as 'position', which is sorted by the serial running time with recursive iterative deepening. A positive residual indicates that the model understates the running time, while a negative residual indicates that the model overstates the running time.

The residual plots show the effect of failing to include various terms.

- Every position indicator variable, except for k_3, (which is shown as position 3 in the 'residuals(Position)' plot), has some datapoints with positive residuals and some with negative

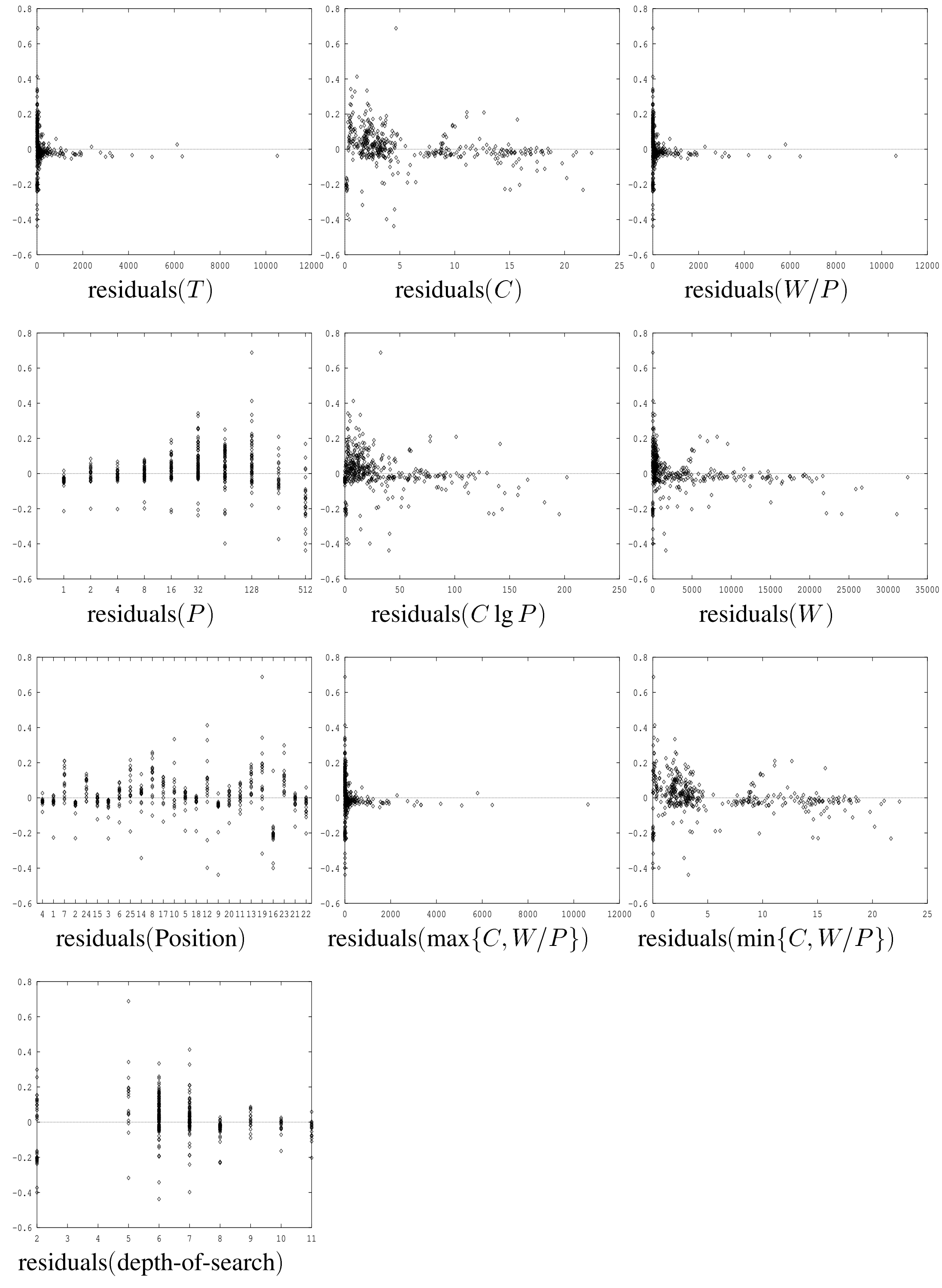


Figure 5-16: Plots of the relative residuals of Equation 5.4 as a function of various quantities: time T, measured critical path length C, total work W, number of processors P, position (sorted by serial running time), and depth-of-search.

residuals. Position k_3 has negative residuals only, but even for k_3 the maximum relative error is only about 22%.

- There is some evidence that the $C \lg P$ term actually may be important. For very small values of $C \lg P$, the residuals tend to be positive, while for the largest values of $C \lg P$ (of which there are not very many data points) tend to have negative residuals. The skew is small, but statistically significant.

- The decision to remove the W term has a similar skew. For small amounts of work, the residuals are positive and for large amounts of measured work, the residuals are negative. The skew as a function of W is not statistically significant, however.

- The residuals as a function of the remaining terms look pretty random.

I tried fitting to a model that included k_3 and $C \lg P$, and obtained

$$\begin{aligned}
\hat{T} &= (3.986 \pm 0.146) + (0.9703 \pm 0.0179)W/P + (3.639 \pm 0.387)C \\
&\quad -(0.264 \pm 0.0465)C \lg P + (15.930 \pm 5.791)k_3, \\
\text{CORR} &= 0.999052, \\
\text{MRE} &= 0.02999, \\
\text{MAXE} &= 0.6621.
\end{aligned}$$

The values of the C and W/P coefficients change quite a bit, which is due to the fact that $C \lg P$ is strongly correlated to both C and W/P. For the largest machine we measured ($P = 512$) the $C \lg P$ term is still dominated by the C term. Further experiments on even larger machines might help to answer the question of whether $C \lg P$ is significant. There is also fairly strong evidence that one of the positions (Position 3) has a 10 to 20 second startup cost.

If we try to simplify the model even more, it no longer accurately models the performance. Getting rid of the W/P term yields

$$\begin{aligned}
\hat{T} &= (3.682 \pm 4.465) + (3.978 \pm 2.647)C, \\
\text{CORR} &= 0.960681, \\
\text{MRE} &= 0.3570, \\
\text{MAXE} &= 1.4696.
\end{aligned}$$

While the sample correlation coefficient is not *terrible*, the mean relative error has jumped up by an order of magnitude, the confidence intervals for the coefficients are very wide, and the residuals as a function of W/P are very skewed (see Figure 5-17.)

Removing the C does not look as bad as removing W/P.

$$\begin{aligned}
\hat{T} &= (5.566 \pm 0.539) + (1.140 \pm 0.0678)W/P, \\
\text{CORR} &= 0.996043, \\
\text{MRE} &= 0.08977, \\
\text{MAXE} &= 0.8462.
\end{aligned}$$

The residuals as a function of P are increasing, however. (See Figure 5-18.) Recall that a positive residual means that the model understates the runtime. Thus, if we do not include the W/P term, for large numbers of processors (where the critical path length is important) the runtime is understated

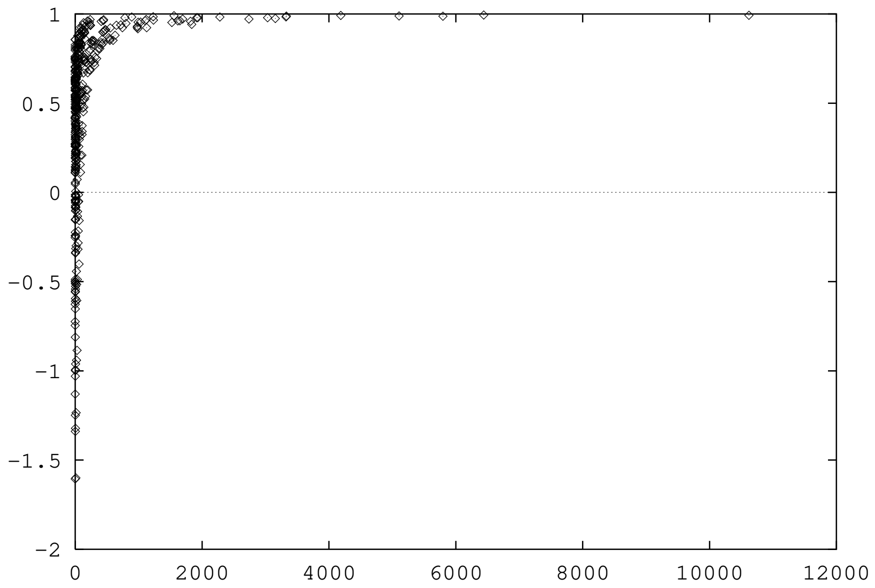


Figure 5-17: Plot of the relative residuals of $3.682 + 3.978C$ as a function of W/P. The residuals are clearly an increasing function of W/P.

and for small numbers of processors the runtime is overstated. To get a good model, we really do need both the C and W/P terms. Those two terms, as shown in Equation 5.34, provide an excellent model of the performance of the system.

Except for the 4.3-second constant term, the StarTech scheduler is within a small factor of optimal. If we take $\hat{T} = 1.5C + 1.02W/P$ as the model, then can conclude that the scheduler is with a factor of 2.52 of optimal, following a two case analysis.

- If $W/P \geq C$, then

$$\begin{aligned} 1.5C + 1.02\frac{W}{P} &\leq 1.5\frac{W}{P} + 1.02\frac{W}{P} \\ &= 2.52\frac{W}{P}. \end{aligned}$$

  Since W/P is a lower bound to the run time, we are within 2.52 of optimal in this case.

- If $W/P < C$ then, similarly

$$\begin{aligned} 1.5C + 1.02\frac{W}{P} &< 1.5C + 1.02C \\ &= 2.52C. \end{aligned}$$

  Since C is a lower bound to the run time, this case is also within a factor of 2.52 of optimal.

The 4.3-second constant term is probably mostly an artifact of the measurement strategy that I used. StarTech, like Hitech, performs several seconds of precomputation on the front-end before starting up the game tree search. For example, during the the program fills in tables that are used for the static evaluation function. My critical path length measurements do not include this

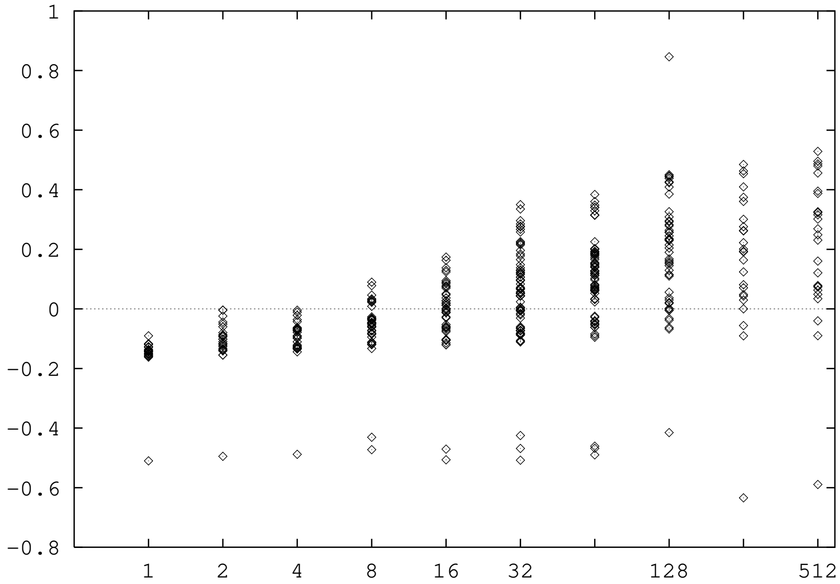


Figure 5-18: Plot of the relative residuals of $5.566 + 1.14W/P$ as a function of P.

precomputation. My run-time measurements, however, do include the precomputation. Since the precomputation is performed serially before the game tree search is started, the precomputation cost should be added to the critical path length. If I had correctly measured the critical path length, the constant term would probably be even smaller.

5.7 Swamping

We have studied the performance of the StarTech scheduler, which includes for example the global throttling mechanism. In this section, we analyze the swamping problem, which leads us to the conclusion that global synchronization can help schedule any dynamic MIMD-style program.

A naive work-stealing scheduler may not be able to guarantee that the computation makes progress. In such a work-stealing scheduler, each of the idle processors sends a request for work to another processor, and if the request is denied, then the idle processor repeats, sending a request to yet another processor. If there is a single busy processor that has no work to spawn, that processor can end up spending all its time dealing with incoming requests. By the time it has dealt with a request, another request may have arrive. In this case, the processor is *swamped* by all the requests for work.

The swamping problem has been reported or alluded to several times in the literature. R. Finkel and U. Manber observed "network flooding" in DIB [FM87, p.243] when the program is near termination. The DIB package solves the problem by introducing constant delays between requests for work. The implementors of Zugzwang chess program observed swamping throughout the search [FM93], resulting in unstable performance. To solve the problem, they tested two strategies that force the stealing processor to wait before sending out another request. They found that for Zugzwang, waiting a constant time between every request works, but it is tricky to pick the right constant. Too small a constant results in swamping, and too large a constant results in poor load balancing. They tried an adaptive strategy, where, after receiving a certain number of "denials", the stealing processor waits for a constant delay. For Zugzwang, the adaptive strategy with constant backoff stabilizes the performance of the program without too much tuning.

To understand under what conditions a processor can be guaranteed to make progress on its local work even in the face of incoming messages, I explored the swamping problem in isolation. I implemented a naive work-stealing tree search program called SWAMP. In the SWAMP program, Processor 0 has about 0.05 seconds of work to do, but there is nothing to steal. Meanwhile all the other processors try to steal work from random processors, but all of their requests to steal work are denied.

The SWAMP program is parameterized by three values:

- t_b is the time for a busy processor to service a request to steal work.
- t_i is the time for an idle processor to service a request to steal work.
- t_s is the time from when a denial message is sent back to a requester to when the requestor's next request message arrives at a processor.

It turns out that the *leverage*

$$z = \frac{t_b}{t_s + t_i} \tag{5.36}$$

is the critical parameter of the swamping program. Figure 5-19 shows the precise relationship between t_b, t_i, and t_s. When a stealing processor (shown in the middle) sends a request to steal, the message eventually arrives at a busy or an idle processor (shown at left). The amount of time that the other processor spends handling the message is called t_b or t_i, depending on whether it is a busy or an idle processor. The response is sent back, the stealing processor receives the denial, and then the stealing processor sends another request to some other processor (at right). We define t_s to be the time from when the first denial was sent to when the second request was received. All of an idle processor's time can be accounted for as being "between requests" or "waiting for a busy or idle processor".

I measured the time it takes for the busy processor to get 0.05 seconds of work done while being interrupted by requests for work. Figure 5-20 shows the situation for a typical set of parameters. In this experiment, I set $t_i = 5$ microseconds, $t_s = 10$ microseconds, and I varied t_b and the number of processors. As t_b grows, the time to run the problem rises dramatically. I set an artificial timeout of 30 seconds (which shows up on the graph at about 29 seconds due to startup overheads.) As the number of processors grows, the rise in the curve comes sooner and faster. For a run on two processors, the curve grows very slowly.

This system can be modeled as a closed Jackson network [Kle76, p. 151], as shown in Figure 5-22. In this model, we have N processors, each modeled by an exponential server with parameter μ_i. In this throttle experiment, the servers are constant-time servers, but it turns out that a model using exponential servers accurately predicts the behavior of the system. Recall that an exponential server with parameter μ has the property that

$$P[\text{service time} \le x] = 1 - e^{\mu x},$$

which means that the average service time is $1/\mu$. In this model, Processor 0, the busy processor, has server parameter

$$\mu_0 = \frac{1}{t_b}$$

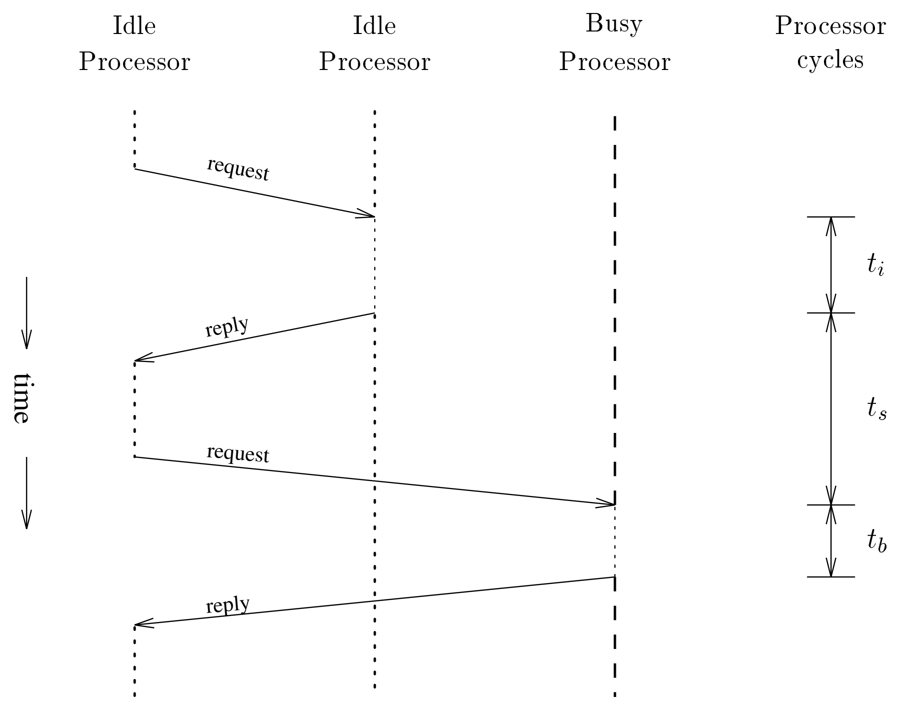


Figure 5-19: The relationship of t_b, t_i and t_s in the SWAMP program. The time required of a busy processor to service a request is t_b. The time for an idle processor to service a request is t_i. The round trip time from when a denial is sent by a processor to back to an idle requesting processor, to the time that the requestor's next request arrives at a processor is t_s.

and all the other processors have

$$\mu_i = \frac{1}{t_i + t_s}.$$

When a customer leaves a server, the customer goes to a randomly chosen server.

Given this setup, the question is, "How long does it take until Processor 0 has not been servicing requests for a total of 0.05 seconds?" This question tells us how long it will take for the processor to get 0.05 seconds worth of work done, which relates understanding what fraction of the time a processor is busy servicing requests instead of getting useful work done.

The state of this system can be represented as a vector $(k_0, \ldots, k_{N-1})$ of the number of customers at each server, that is, there are are k_i customers at server i. For a closed system, the probability of being in any particular state can be solved by first solving the following set of linear equations for x_i:

$$\mu_i x_i = \sum_{j=0}^{N-1} \mu_j x_j r_{ji},$$

where r_{ji} is the probability that a customer leaving server j proceeds to server i. There are only $N-1$ independent equations, and the values of the x_i's are only determined to within a constant. Thus, all that really matters is the ratios x_i/x_j. Jackson showed that for such a network of queues,

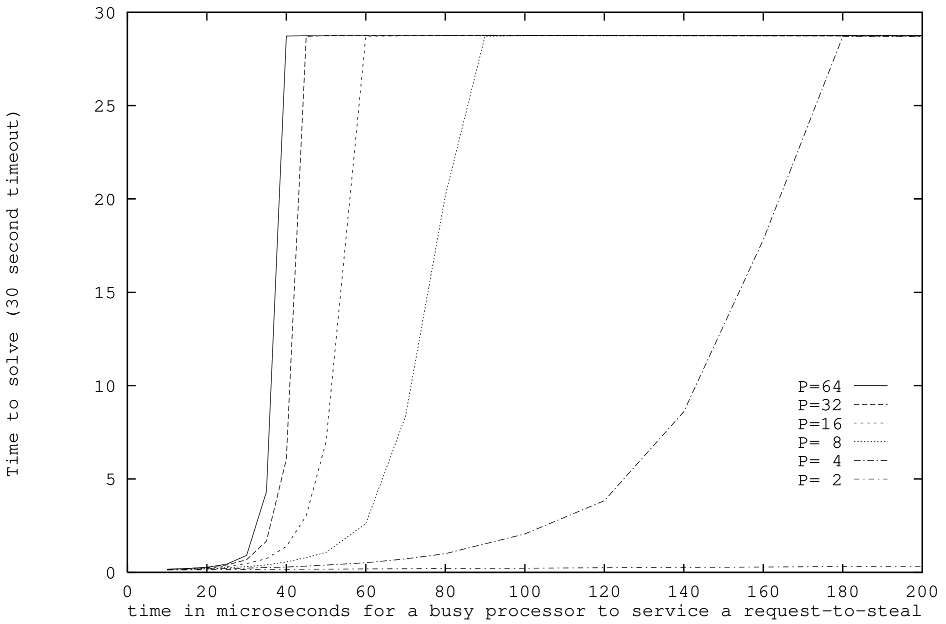


Figure 5-20: Results for the isolated swamping experiment varying the number of processors and t_b, with $t_i = 1$ microseconds and $t_s = 10$ microseconds.

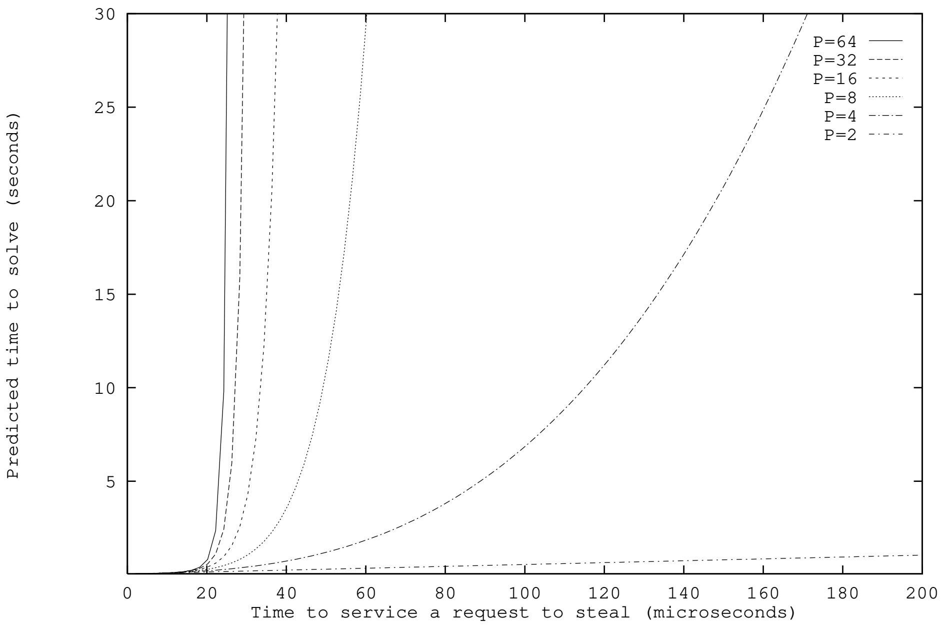


Figure 5-21: The analytic queueing model for the swamping experiment.

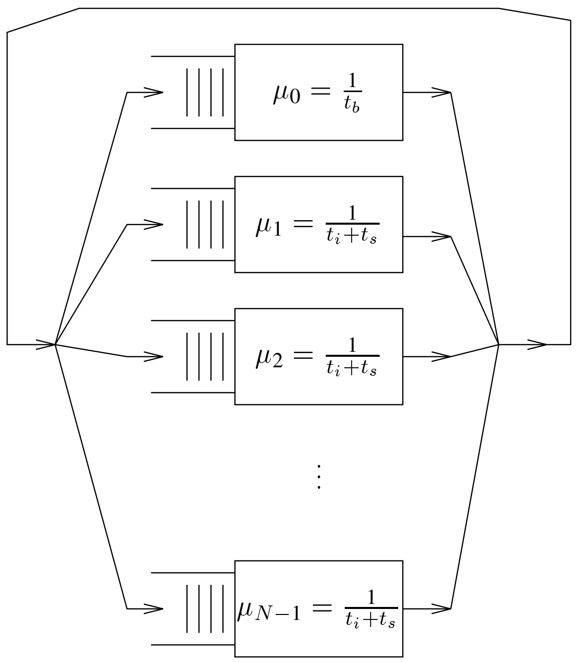


Figure 5-22: This Jackson network models the swamping problem. The critical parameter is the *leverage* $z = t_b/(t_s + t_i)$.

each with a single exponential server, the probability of the system being in a given state is

$$p_{(k_0,\ldots,k_{N-1})} = \frac{1}{G(K)} \prod_{i=0}^{N-1} x_i^{k_i},$$

where the normalization constant is

$$G(K) = \sum_{(k_0,\ldots,k_{N-1})\in A} \prod_{i=0}^{N-1} x_i^{k_0}$$

and the set of all possible state vectors is

$$A = \left\{ (k_0, \ldots, k_{N-1}) : k_i >= 0, \sum_{i=0}^{N-1} = N-1 \right\}.$$

For the system of Figure 5-22, we have $r_{ij} = 1/N$. Without loss of generality we set $x_i = 1$ for $i > 0$, since $x_i = x_j$ for $i, j > 0$ and only the ratios of the x_i's matter. Thus, the remaining variable to be solved is x_0. The probability of being in a particular state is

$$p_{(k_0,\ldots,k_{N-1})} = \frac{z^{k_0}}{G(K)},$$

where

$$G(K) = \sum_{(k_0,\ldots,k_{N-1})\in A} z^{k_0}.$$

If we define

$$(i,*) = \{(k_0,\ldots,k_{N-1}) \in A : k_0 = i\}, \tag{5.37}$$

then we can rephrase the question as "What is the probability that we are in $(0,*)$?" The states $(0,*)$ are precisely the states in which Server 0 is not servicing any customer. Observing that for any element $\vec{k} \in (0,*)$, we have

$$\begin{aligned} p_{\vec{k}} &= \frac{z^{k_0}}{G(K)} \\ &= \frac{1}{G(K)}, \end{aligned}$$

and thus,

$$\begin{aligned} p_{(0,*)} &= \sum_{\vec{k}\in(0,*)} p_{\vec{k}} \\ &= \sum_{\vec{k}\in(0,*)} \frac{1}{G(K)} \\ &= \frac{|(0,*)|}{G(K)}. \end{aligned}$$

The cardinality of $(0,*)$ is the number of ways to add up $(N-1)$ nonnegative integers to get $(N-1)$. In general the number of ways to add up N nonnegative integers to get M for $M \leq N$ is

$$\mathcal{N}_{N,M} = \begin{pmatrix} N+M-1 \\ M-1 \end{pmatrix}, \tag{5.38}$$

and so

$$|(0,*)| = \begin{pmatrix} 2N-3 \\ N-2 \end{pmatrix},$$

and

$$\begin{aligned} G(K) &= \sum_{(k_0,\ldots,k_{N-1})\in A} z^{k_0} \\ &= \sum_{k_0=0}^{N-1}\left(\sum_{(k_0,k_1,\ldots,k_{N-1})\in A} z^{k_0}\right) \\ &= \sum_{k_0=0}^{N-1}\left(z^{k_0}\left(\sum_{(k_0,k_1,\ldots,k_{N-1})\in A} 1\right)\right) \\ &= \sum_{k_0=0}^{N-1}\left(z^{k_0}|(k_0,*)|\right). \end{aligned}$$

The cardinality of $(k_0,*)$ is the number of ways to add up $(N-1)$ nonnegative integers to get

$(N - k_0 - 1)$, so using Equation 5.38 again,

$$G(K) = \sum_{i=0}^{N-1} \left(z^i \left(\begin{array}{c} 2N - i - 3 \\ N - 2 \end{array} \right) \right),$$

yielding

$$p_{(0,*)} = \frac{\left(\begin{array}{c} 2N - 3 \\ N - 2 \end{array} \right)}{\sum_{i=0}^{N-1} \left(\begin{array}{c} 2N - i - 3 \\ N - 2 \end{array} \right) z^i}. \tag{5.39}$$

The answer to our question is that the time it takes for the processor to get w seconds worth of work done is

$$t_w = \frac{w}{p_{(0,*)}}. \tag{5.40}$$

Figure 5-23 shows the values for $p_{(0,*)}$ expanded for a few small values of N. Plugging the values from the experimental setup into the model, we get the graph shown in Figure 5-21, which qualitatively matches the observed behavior of the experimental setup. As the number of processors N increases, the likelihood of Processor 0 getting any work done drops dramatically as a function of the leverage z. Considering just the z^{N-1} term of Equation 5.40 we have

$$\begin{aligned} t_w &= w \frac{z^{N-1} + \ldots}{\left(\begin{array}{c} 2N - 3 \\ N - 2 \end{array} \right)} \\ &> \frac{wz^{N-1}}{\left(\begin{array}{c} 2N - 3 \\ N - 2 \end{array} \right)} \\ &= \frac{wz^{N-1}(N-2)!(N-1)!}{(2N-3)!}. \end{aligned}$$

Using Stirling's approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \tag{5.41}$$

and simplifying

$$\begin{aligned} t_w &> wz^{N-1} \sqrt{\frac{2\pi(N-1)(N-2)}{2N-3}} \frac{(N-2)^{N-2}(N-1)^{N-1}}{(2N-3)^{2N-3}} \\ &\approx w\sqrt{\pi N} \frac{z^{N-1}}{2^{2N-3}} \\ &= 2w\sqrt{\pi N} \frac{z^{N-1}}{4^{N-1}} \\ &= 2w\sqrt{\pi N} \left(\frac{z}{4} \right)^{N-1}. \end{aligned}$$

Thus, for N processors the rate at which work gets done drops off at least exponentially with N.

This phenomenon has not been systematically studied before, probably because most other

N	$p_{(0,*)}$
2	$1/(z+1)$
3	$3/(z^2+2z+3)$
4	$10/(z^3+3z^2+6z+10)$
8	$1716/(z^7+7z^6+28z^5+84z^4+210z^3+462z^2+924z+1716)$

Figure 5-23: The value of $p_{(0,*)}$, the probability that at a given time a given processor is making progress on its own work, expanded for a few different machine sizes. For large numbers of processors, the probability is exponentially decreasing with the leverage.

researchers have used relatively small machines. It takes a relatively large leverage for swamping to become significant on small machines, and as soon as the application finds enough parallelism to keep all the processors busy, the swamping issue goes away. Looking at only a small machine, it would not be evident that a naive work-stealing protocol can have a severe scaling problem.

Why might the leverage be $z > 1$? The leverage is essentially the ratio of the time taken by a busy processor to respond to a request to the time taken by an idle processor to a request. The idle processor can clearly respond quickly since it has nothing else to do. The busy processor takes longer due to the time to context switch from the busy work to the request handler.[9]

The model of Equation 5.40 does not exactly match the experimental results of Figure 5-20. There are many possible reasons that the model is a little bit inaccurate, besides the exponential server assumption and the fact that congestion in the CM-5 data network introduces additional queueing delays. The easiest way to further validate this model of swamping would be to perform a simulation study that more closely matches the model. For example, it might be worthwhile to examine the swamping problem on an ethernet, which is much easier to model accurately with queueing theory than is the CM-5 data network, and to use exponential servers instead of constant time servers. If the model presented here does completely explain the swamping problem on the CM-5, one should be able to get an arbitrarily close match to the actual performance of such a simplified system. Even if there are other effects in the actual swamping problem, however, this model serves as a convincing explanation of why swamping gets so bad so quickly. We have learned that to avoid the swamping problem for work-stealing schedulers we can either try to keep the leverage small, to run on only a few processors, or to use some sort of throttle to change the underlying behavior. To solve the swamping problem for StarTech, I implemented a global throttle that uses the global synchronization of the CM-5 (see Chapter 4).

Using a global throttle is not the only way to solve the the swamping problem. One approach to solving this problem is to schedule the processor fairly between servicing the requests to steal and the local work to be done. The disadvantage of such an approach is that the data network might start clogging up with unhandled request messages. Even in a circuit switched network, which forces the processor to retry if a message collides, the performance of the system might degrade as blocked processors that are trying to send requests themselves fail to respond to other processors' requests, causing all of the processors to slow down. In Chapter 3 we saw that barrier synchronization is an important global end-to-end flow-control mechanism for bulk data transfers as well.

[9]In a polled message-passing system, the request message may have to wait for about half a thread length before it is handled. This waiting time does not directly contribute to swamping on the busy processor, since during that wait time the processor actually gets some work done. This delay may slow down the system in other ways, however, such as by congesting the data network.

One way to keep the swamping problem small is to keep the leverage z small. For example, the Alewife processor [ACD*91] can fetch data out of a single word of the processor's memory without disturbing the processor. A busy processor can set up a single word to hold the amount of work that is available to steal. If there is work to steal, the processor does not mind spending some time distributing it. If there is no work to steal, the processor is allowed to do its work undisturbed, while the potential thieves quickly determine that there is nothing to steal. This approach argues for providing a class of memory-access messages that are handled quickly and efficiently by the interface between the processor and the network. The leverage can be kept small by careful hardware or software design. Such strategies require careful tuning, My global throttling strategy, on the other hand, is robust, needs very little tuning (see section 5.7), and costs very little (see section 6.6).

Another approach is to use an adaptive randomized backoff strategy, such as is found in Ethernet [MB76]. Such backoff strategies are easier to tune than trying to adjust the leverage, but they require more tuning than does my global throttling strategy. DIB uses an adaptive backoff strategy (as opposed to a random backoff strategy) [FM87]. PCM uses a randomized backoff strategy [HZJ94, Hal94].

All of these approaches take advantage of globally consistent time to solve the swamping problem. The fair schedulers guarantee that a certain fraction of the real time spent at a processor is doing work. The adaptive backoff strategy takes advantage of a real clock. StarTech's global throttle strategy puts a thin veneer of global real time on top of the otherwise asynchronous computation.

5.8 A Space-Time Tradeoff

Now that we understand the performance of the StarTech scheduler, we consider the ramifications of a decision made to get a good space bound. StarTech maintains an invariant that guarantees that it will not run out of memory. Each activation frame has a *height*, measured as the distance from the frame to the root of the computation. StarTech guarantees that on any given processor there is at most one frame of any depth.

The memory requirements per processor are thus similar to the memory requirements for a serial game-tree searching program. StarTech's activation frames are a little bit larger than the activation frames used for a serial computation, since the parallel frame keeps information for each of the children, while the serial frame only needs to keep information about one child at a time. The frames are only a little bit larger however. Thus, the memory requirements per processor, of StarTech, are about the same as the memory requirements for a serial computation, which also needs only one activation frame of any depth.[10]

To maintain StarTech's invariant requires some care, since StarTech does not move activation frames from one processor to another. Consider the situation in which a processor has had a single child stolen, but otherwise has no work to do. It is waiting for the child to return its result. If the processor sits idle, then processor cycles are wasted. If the processor steals work, then it might end up with more than one frame at the same depth, and eventually run out of frame memory. To maintain the invariant, StarTech uses a simple rule. A processor may not steal work until all of its local frames have been completed. As a consequence, in our example, the processor sits idle waiting for the result from its child.

The price paid for StarTech's simple rule is at worst-case a factor of H slowdown, where H is the height of the tree. At any point during the dynamic unfolding of the tree, one always can do work at any leaf. In any tree of height H with l leaves, there are no more than $H \cdot l$ internal positions

[10]Robert D. Blumofe showed me this explanation of StarTech's space bounds.

(and that worst case happens only on very tall skinny trees). Therefore, we lose at most a factor of H from this decision. The ratio of internal frames to leaves is less than H for the shallow bushy trees typically searched in a chess program.

There are ways to avoid this worst-case slowdown. The Zugzwang chess program [FMM92], tries to avoid even this small performance penalty by allowing a stalling processor to steal work from one of its children. Feldmann refers to this as the *helpful master* strategy, and while such a strategy should not hurt the performance, it does not reduce the worst-case bound. Blumofe and Leiserson [BL93] observe that by moving frames from one processor to another, the worst case can be substantially improved. Another approach would be to use on-processor multithreading to allow up to K frames per processor per level in the call tree. The worst-case bound does not get much better, but the typical case might show some improvement.

Several systems to support dynamic MIMD-style programming have been proposed and implemented [DR81, Hal84, Bir89, SYH*89, PC90, NPA92]. Those systems do not provide predictable, high-performance nor do they provide any space bounds on running programs. D. Culler provides some *ad hoc* techniques for managing the space bounds of a dataflow program [Cul89]. By limiting the number of iterations of a parallel loop that can run at the same time, Culler manages to control the space bounds of data-parallel programs written in a dataflow language. Culler's techniques do not generalize to arbitrary dataflow or dynamic-MIMD programs.

How much performance could we hope to actually get back by changing StarTech's simple mechanism to something more complex? My measurements indicate that between 1% and 3% of the processor cycles are used up waiting for children. Furthermore, in the performance model

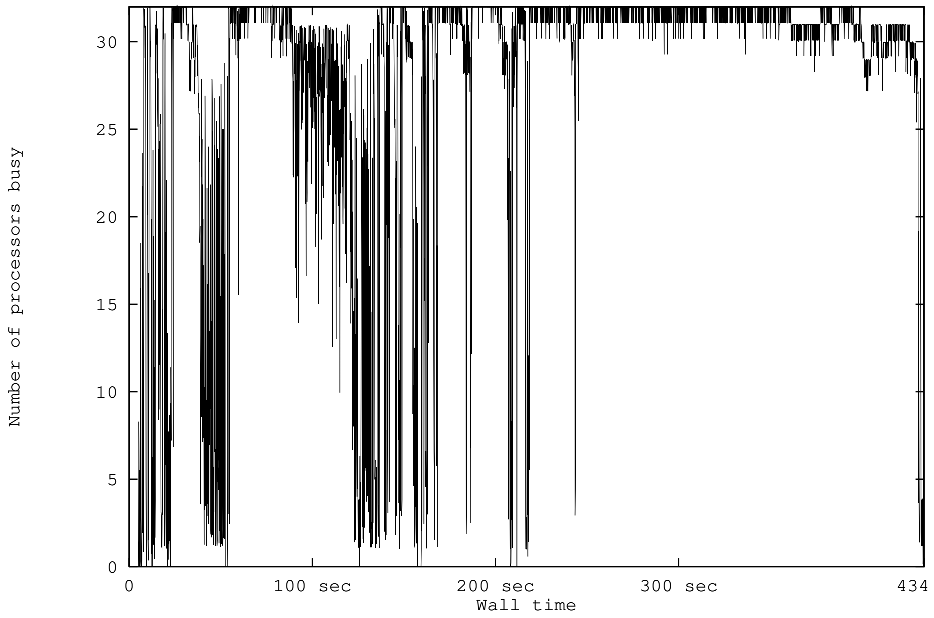
$$\hat{T} \approx 1.02\frac{W}{P} + 1.5C + 4.3 \text{ seconds},$$

there is only a 2% overhead added to the linear speedup term W/P. Since, in my performance measurements, I did not count the time waiting on children as work, it is possible that all of the 2% overhead is due to waiting on children.

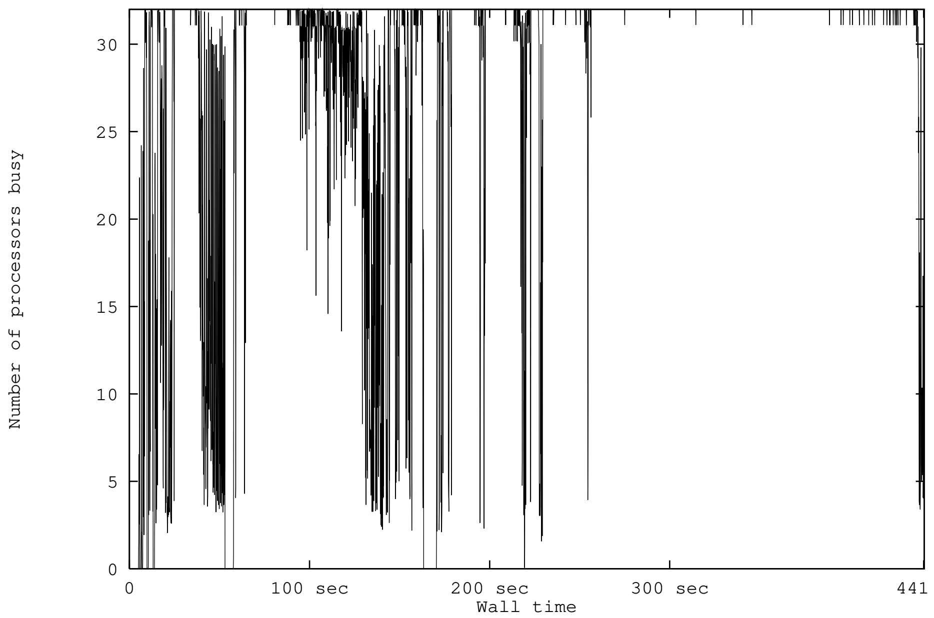
Another way to see how much time is spent waiting on children is to consider the processor utilization profile of Section 5.5. The profile is reproduced here as Figure 5-24(a) without shading in the region under the curve to make it easier to see what is going on when there is plenty of parallelism. For most of the second half of the profile, in the processor-saturation region, the processors stay almost completely busy. But the utilization jumps around between 30, 31, and 32 processors. For example, whenever the utilization is only 30, there is a 6% inefficiency. If instead of counting the time spent waiting on a child as idle time, we were to count it as work-time, we should see the work increase. Figure 5-24(b) shows the utilization profile when the time waiting on child is accounted for as work. In this case the utilization is nearly 100% during the processor-saturation region. The difference between the two profiles shows the distribution of the time spent waiting on children.

There are at two reasons that during the highly parallel parts of the program, the processor utilization is not pegged at 100%.

- StarTech uses a randomized work stealing scheduler. When a processor is idle, it requests work from another, random, processor. Cycles can be lost just looking for work that, in principle, could be worked on immediately.
- The StarTech scheduler guarantees a space bound by waiting on a child. In principle the work of the position that is waiting on a child could be moved to the child, and then the processor could do something useful. It turns out that this is causing the inefficiency that we can see in the utilization profile.



(a) Correct accounting of child-wait time.



(b) Treat child-wait time as busy-time.

Figure 5-24: The inefficiency during the saturation region is mostly accounted for by the time waiting on children. Graph (a), a reproduction of Figure 5-15 without the region under the curve shaded, shows the processor utilization when child-wait time is accounted for correctly as idle time. Graph (b) shows how the apparent efficiency improves if we account for the time waiting for children as time that the processors are doing useful work. Most of the inefficiency in the saturation region goes way when we ignore the child-wait time.

It appears that most of the inefficiency is actually due to the layout decision rather than to StarTech's randomized scheduling strategy. Potentially that 2% overhead could be redeemed by adopting a different strategy for achieving the space bound.

Chapter 6

Tuning the StarTech Program

6.1 Introduction

As we have seen, the critical path length and total work give us insight into how long our program will run on a given size machine. This chapter explores how to use this insight to help speed up the program. I tried several techniques for improving the program, including using recursive iterative deepening to improve the move ordering, using I-structure-like data structures for improving the performance of the search for reconverging subtrees, and modifying the basic Jamboree algorithm to reduce the work.

This chapter starts with a discussion in Section 6.2 of how to measure the performance of parallel chess programs. Section 6.3 explains the implementation of StarTech's global transposition table. Section 6.4 studies how the transposition table can be made more effective, especially for a parallel chess program. Section 6.5 studies modifications to the Jamboree algorithm that improve the work efficiency, hopefully without increasing the critical path length too much. Finally Section 6.6 shows where the processor cycles are spent when running StarTech.

6.2 What is the Right Way to Measure Speedup of a Chess Program?

The performance of any chess player can be determined by a system due to Elo [Elo78], which rates players according to their performance in tournament play.[1] Such a scheme is time consuming and impractical for measuring the improvement due to changes in a chess program. Instead I use a set of benchmark problems designed by International Master L. Kaufman [Kau92, Kau93] to determine how a change in my program affects the quality of play.

To obtain an estimated Elo rating for a program, Kaufman uses 25 chess positions (20 tactical, 5 positional), each of which has a correct answer. To obtain an estimated rating, one measures the time it takes for the program to find Kaufman's correct answer for each position. Then, one throws away the worst 5 times and sums up the remaining times. Let t_{20} be the sum of the times, in

[1] In the Elo system, if two players with a 200 point ratings difference play a game of chess, the game has an expected value of 0.75 points for the higher ranked player, where a win is worth 1 point, a draw is worth 0.5 points, and a loss is worth 0 points. Thus, it is the *difference* between the ratings of two players that is important. The absolute value of the Elo ratings of a group of players varies according to an arbitrary decision of what some particular player's rating is. Here we use the USCF (United States Chess Federation) rating scale, which sets the rating of the 'average rated member' of the USCF at 1500 USCF.

seconds, to solve the fastest 20 positions. Given t_{20}, Kaufman estimates the USCF rating as

$$\text{USCF Rating} \approx 2930 - 200 \cdot \log_{10} t_{20} \tag{6.1}$$

I.e., a factor of 10 in performance is estimated to be worth 200 ratings points, or about one standard deviation in the population of rated chess players. Kaufman has chosen a particular way of weighting the different positions, and there are other ways to weight positions, such as measuring the depth of search for a given amount of time. Kaufman's estimator is simple and provides well-defined way to compare different chess programs. Kaufman cautions against misuse of his ratings estimator, which is designed only to predict accurately ratings of programs that are near the range of a chess master or Senior Master (2200–2600 USCF).

I measured my parallel implementation against my best serial implementation of StarTech. Our serial implementation of StarTech uses standard α-β search[2], and some significant effort was put into making the serial program run as fast as possible (partly because much of the code is shared so such investment pays off for the parallel program too, and partly because I wanted to have the best serial program I could get). I ran the serial program on a Sun workstation,[3] which has a larger cache, a better cache controller, more main memory, and a faster processor than each of the CM-5 node processors. The main advantage of the Sun workstation over the CM-5 nodes that I used elsewhere is that it can have a larger transposition table.

I wanted to measure the improved rating of StarTech as a function of the number of processors, but first I had to isolate other factors. The biggest other factor is the effect of the transposition-table size which varies with the number of processors.

Program StarTech, like most chess programs, uses a *transposition table* to cache results of recent searches. For a search from a given position to a certain depth, the transposition table indicates the value of the position and the best move for that position. The transposition table is indexed by a hash key derived from the position. Whenever the search routine finishes with a position, it modifies the transposition table by writing the value back (it may decide that the old value stored was better to keep than the new value.) Whenever the search routine examines a position, the routine first checks to see if the position's value has been stored in the transposition table. If the value is present, then the routine can simply return the value. Sometimes, the value for the position is not present, but a best move is present for a search to a shallower depth. In this case the best move for the shallow depth can be used to improve the move ordering. Since the Jamboree search algorithm depends on good move ordering, the transposition table is very important to the performance of StarTech.

Hsu argues [Hsu90] that if one increases the size of the transposition table along with the number of processors, then the results are suspect. Hsu states that increasing the transposition table size by a factor of 256 can easily improve the performance by a factor of 2 to 5. Our strategy is to choose a transposition table size that is sufficiently large that increasing it further doesn't help the performance. Figure 6-1 shows the estimated rating of the serial program as a function of the transposition table size, and it also shows the number of positions visited by the program under each configuration. In our serial implementation, larger transposition tables take longer to initialize, but I did not count the cost of initializing the larger transposition table against the serial program.

Note that the number of positions visited by the search tree monotonically decreases as the table gets larger, but that after 2^{23} entries, the number of positions becomes constant. We can conclude

[2] I really ought to have used a serial Scout search, which is typically worth an additional 5–10% performance improvement.

[3] The workstation is a four-processor Sun model S690-140-128-P56. The machine was unloaded except for my job and whatever time was spent servicing the ethernet and handling operating system overheads.

Transposition Table Entries	Positions Visited	Time (seconds) top 20	Estimated Rating
0	161,625,376	23337.14	2056
2^{16}	94,409,196	13506.36	2104
2^{17}	85,753,262	12670.01	2109
2^{18}	76,498,040	10925.46	2122
2^{19}	65,568,814	9605.36	2133
2^{20}	55,910,651	8040.08	2149
2^{21}	48,256,980	7138.08	2159
2^{22}	42,627,585	5799.31	2177
2^{23}	40,805,974	6120.18	2173
2^{24}	40,805,974	6364.99	2169

Figure 6-1: Performance of my best serial implementation of StarTech as a function of transposition table size. The number of chess positions in the search tree is shown, along with the time in seconds, and, the estimated rating using Kaufman's ratings estimation function, given by Equation 6.1.

that for Kaufman's ratings test any transposition table size of more than 2^{23} entries is quite sufficient, and a larger transposition table will not, by itself, raise the estimated rating of the program.

I believe that the slight decrease in estimated rating (i.e., the increase in time to solve the problems) beyond 2^{23} entries is due to paging and cache effects, because the machine I ran these tests on could not reliably hold the working set in main memory when the transposition table is larger than 2^{23} entries. Any transposition table of size 2^{22} entries or smaller easily fit within the main memory of the serial computer I used.

I ran Kaufman's test on a variety of different CM-5 configurations. The transposition table size was set at 2^{21} entries per processor, which is the largest size that fits in the 32 Megabyte memory of the CM-5 processors. For runs on fewer than 32 processors, we actually used a 32-processor machine with some processors 'disabled'. In this case, I used the entire distributed memory of the 32-processor machine to implement the global transposition table. As a result, in every parallel run, the transposition table contains a total of at least 2^{26} entries.

Figure 6-2 shows the estimated rating of Startech as a function of the number of processors. According to Kaufman's test, there is a diminishing return as the number of processors increases when only the fastest 20 problems are considered. If we consider the time to solve all 25 problems, however, there are still significant performance gains being made even when moving from a 256-node CM-5 to a 512-node CM-5.

I wondered if Kaufman's test has enough parallelism to show StarTech's strengths to full advantage. On the 512-processor run, for many of the situations, the program spent only a few seconds on a position, and on average the time spent on the fastest 20 moves is only 16 seconds—less than a tenth of the time allowed under tournament time conditions (roughly 180 seconds per move.) Kaufman's problem is a fixed-size problem with a fixed amount of parallelism. I had noticed in the tournament that StarTech's performance, measured in positions per second, is generally much better in the second 90 seconds of a search than during the first 90 seconds of search, and I observed that the positions in Kaufman's test that achieved the best speedup were often discarded by Kaufman's evaluation scheme because they were among the slowest positions.

So I decided to try letting StarTech run under tournament time-conditions with 512 processing

Processors	Time for Top 20 (seconds)	Estimated Elo Rating (USCF)	Time for all (seconds)
1	8936.95	2139	38261.91
2	5376.45	2183	22007.46
4	3152.54	2230	11614.43
8	1932.27	2272	7411.54
16	1240.72	2311	4398.32
32	844.00	2344	2803.33
64	573.19	2378	1670.29
128	444.78	2400	1129.68
256	378.72	2414	907.24
512	319.38	2429	677.11

Figure 6-2: The estimated rating of our parallel implementation of Startech as a function of the number of processors. The time to solve the fastest 20 of Kaufman's test problems is shown, along with the estimated rating (computed with Equation 6.1), and the time to solve all 25 positions.

nodes, allocating about 3 minutes per position. I then determined the depth of search reached by the program. With this information I ran the serial program to the same depth to determine how much "serial-work" StarTech managed to do. (That is, the quality of the choice made by StarTech is the same as the quality of the longer serial search.) Thus, for this experiment, I am not measuring speedup, but slowdown. The biggest slowdown was 480-fold, and most slowdowns were greater than 200-fold on the 512-node machine.

The slowdown experiment is not as well controlled as the speedup experiment. I cannot run the serial program with as big a hash table as the parallel program uses, and my budget for machine time would not allow me to run a serial implementation using the entire memory of a 512 node CM-5. If we accept Hsu's estimate, we should derate our slowdown numbers by up to a factor of 5, especially on the longer runs. Even with this derating, the slowdown of the serial program compared to 512-node StarTech is usually more than 50.

I found that by letting the program search deeper, the linear speedup term grew more quickly than did the critical path, and the overall speedup is improved.

In summary, my experiments do not provide a clear measure of the performance of StarTech on 512 nodes under tournament time controls. The authors of the Zugzwang chess program [FMM93] found that when searching 'easy' positions to a very deep depth, more speedup is achieved than can realistically be expected under tournament conditions. On the other hand, searching the easy problems to a shallow depth does not give the program an opportunity to find parallelism. An effort needs to be made to find harder problems to test parallel programs.

6.3 The Global Transposition Table

Program StarTech uses a *global* transposition table, distributed across all the nodes of the machine, as shown in Figure 6-3. To access the transposition table, which requires communicating from one node to another, an active message protocol is used. The hash key used to index the table is divided into two parts: a processor number and a memory offset. When a frame needs to look up a position in the table, its processor sends an active message to the processor named by the hash key, and that processor responds with a message containing the contents of the table entry.

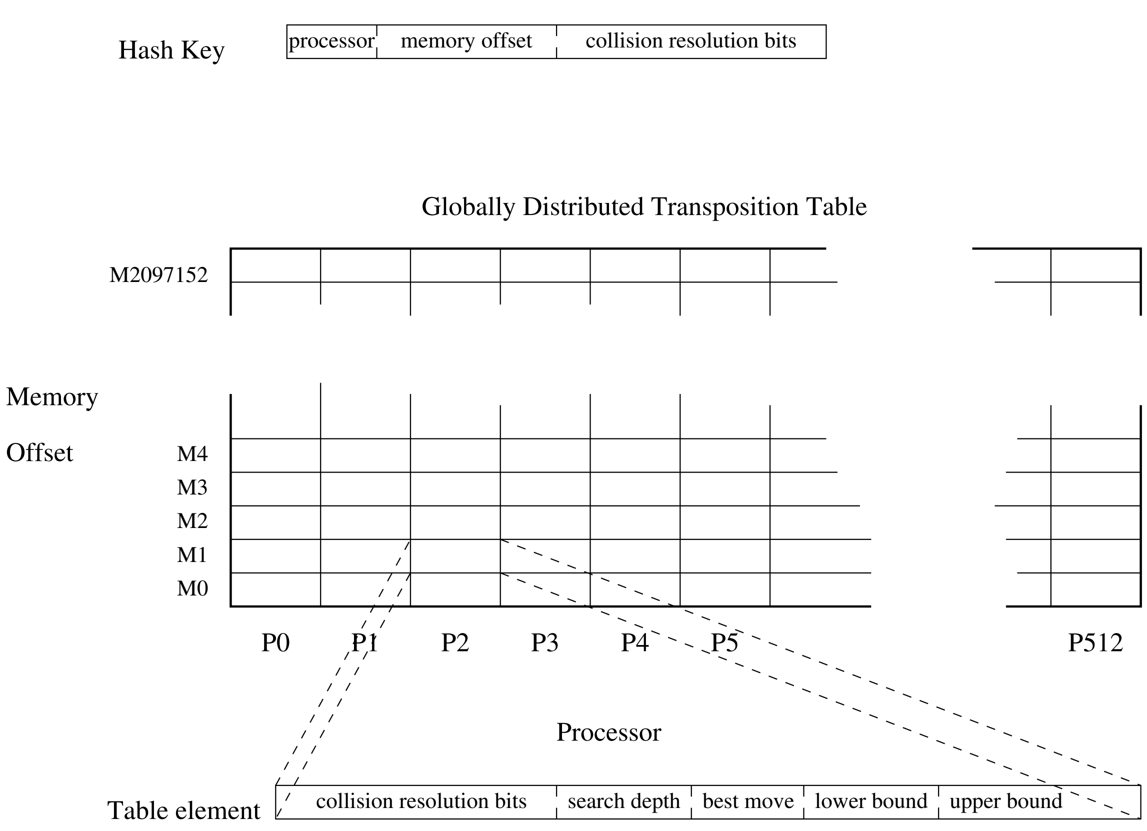


Figure 6-3: The Startech transposition table is globally distributed. The hash key for a position is divided into a processor number, memory offset, and collision resolution bits. The processor number and memory offset identify a unique table element in the machine. The collision resolution bits discriminate between positions that map to the same table element. The table element contains the depth to which the tree was searched, the best move, a lower bound to the value of the position, and an upper bound to the value of the position.

The transposition table is a cache, not a hash table, although the computer-chess literature usually refers to the "transposition hash table".[4] Several different positions can be mapped to the same table entry, and some replacement policy (such as least-recently-used) is used to decide which values to keep and which values to kick out of the table. Most programs, including StarTech, use a direct-mapped strategy, in which each position can be mapped to exactly one table entry. Some other programs, such as Cray-Blitz [HSN89], use a set-associative cache for their transposition table.

We must be careful about providing atomic access to the transposition table. If one processor were to read the first half of a transposition table entry with one message and then read the second half of the transposition table entry with another message, and if a second processor between the two halves of the read issues a write to the transposition table, the first processor receives corrupted

[4]Hash tables and caches both are used to store a collection of objects so that they can be later found. Both use a hash key to determine where any particular object resides in the table. They differ on their replacement policies. When two objects have the same hash key (that is, when they *collide*) in a hash table, some other place is found to put one of the objects. In a cache, when two objects collide, one of the objects is simply thrown away.

data. Similarly, nonatomic write operations can corrupt the data stored in the table. The StarTech transposition-table update operation is, in fact, a read-modify-write. When we have a new value to write, we perform a read and decide which value to keep (the old or new value), and then write back the correct value. We must arrange to perform read and update operations atomically.

There are several ways to implement atomic access to a global data structure. The traditional way is to provide a lock to prevent other processors from accessing the table element. Thus, a processor accessing the element acquires the lock, performs the operations on the element, and then releases the lock. This protocol is inefficient because it requires several round trips of communication through the network.

StarTech uses an optimization that works well for transposition tables, and which gets rid of all the messages except for those messages actually needed to communicate data. StarTech uses active messages [vCG*92] for interprocessor communication. When a message arrives at a processor, a message handler specified in the message is invoked. The handler is run atomically. Thus, to perform the read, we send a "read" message, and the handler reads the value from the hash table atomically and sends it back to the requester. To perform an update we package up all of the information needed to do the update, and send it to the processor that owns the table entry. The message handler does the read, decides what the data should be, and writes the new data, all in an atomic operation. The transformation from a complex locking protocol to the simpler protocol using atomic message handlers can be viewed as a code transformation in which the processor that does the atomic operation is chosen to minimize the amount of locking traffic, as shown in Figure 6-4. This type of transformation may be useful for the simplification of other protocols as well.

We must also be careful that the frame does not get deallocated between the time that the request to the transposition table is sent, and the time that the reply is received back at the frame. Recall, that the frame could be deallocated if, for instance, its parent sends it an "abort" message for some reason. If the reply is received and the frame has been deallocated in the meantime, the entire computation could become corrupted.

This *dangling reference* problem is handled in StarTech by counting messages. The frame records that a message is expected from the global transposition table, and the frame's deallocation becomes dependent on the arrival of that message. This mechanism is essentially the same one as the one described in Section 4.6 that counts the number of messages to arrive before the frame is deallocated.

A similar transposition table scheme is used by the Zugzwang parallel chess program [FMM93], but the issue is not approached from the point of view of obtaining atomic transactions, and the literature on Zugzwang does not make it clear how dangling references are avoided.

F. Popowich and T. Marsland concluded that local transposition tables are better than global transposition tables [PM83]. Local transposition tables do not incur any message passing overhead, but local transposition tables have a much lower hit rate than global transposition tables. With message passing overheads that measure in the tens of milliseconds, Popowich and Marsland were forced to choose between bad performance due to message-passing costs, or bad performance due to poor transposition table effectiveness. The decision is much easier for StarTech, which uses low-overhead active messages on the CM-5.

The Orca system [BKT92] encourages programmers to use a related strategy to obtain atomic access to a data structure. In Orca, the handlers are not run atomically, and there is no constraint on what kind of code can run in a handler. Thus, the natural way to handle atomic access to a data structure is to write a handler that obtains a lock, then accesses the data, and then releases the lock. Traditionally, the overhead of a message passing system that can handle this sort of general remote

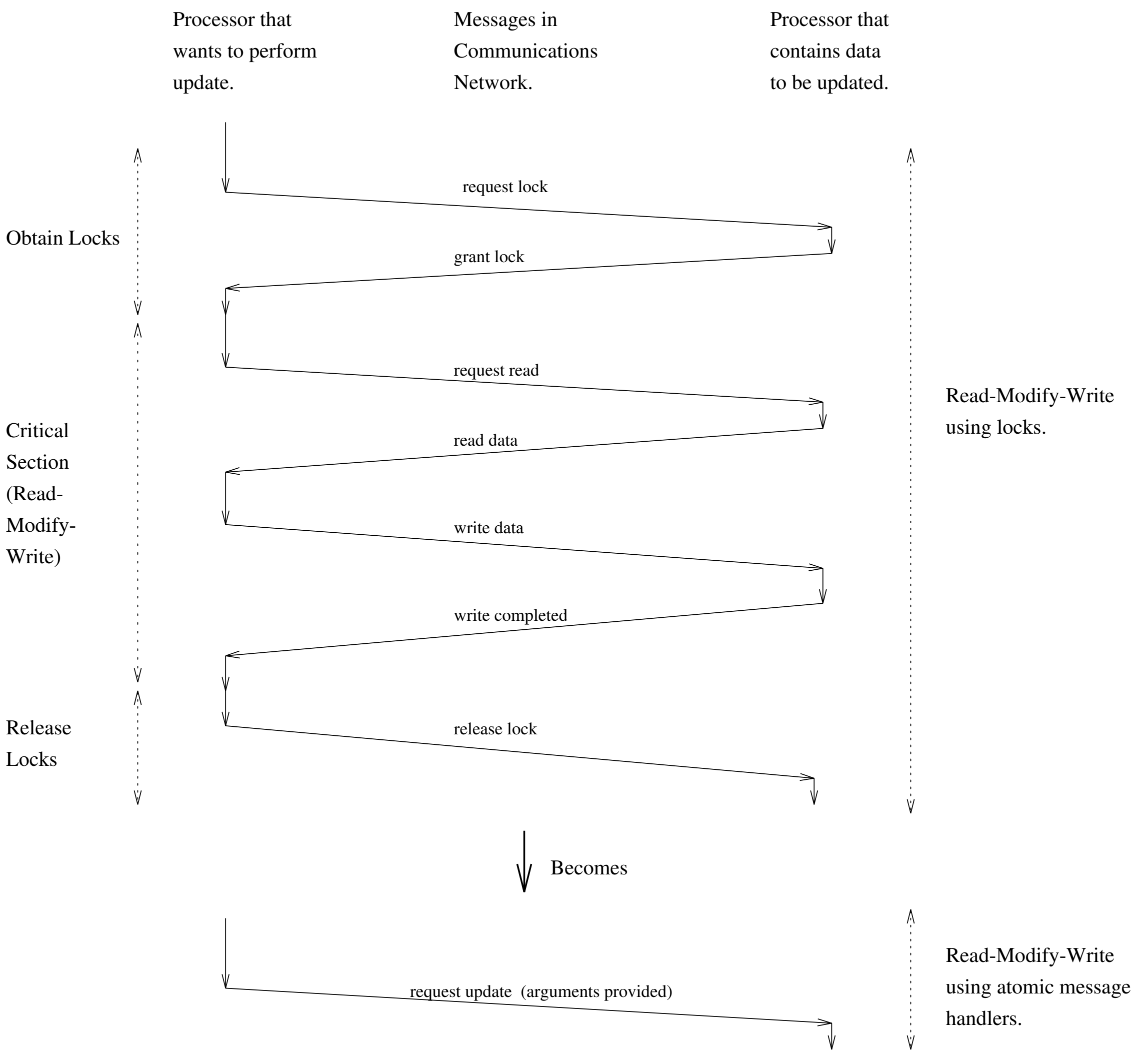


Figure 6-4: How to transform an atomic read-modify-write written with locks into an atomic read-modify-write written using atomic message message handlers, as provided by the active message abstraction. To perform a read-modify-write with locks requires many messages: a round trip to obtain the lock, a round trip to read the data, a round trip to write the data, and then a message to release the lock. The active message implementation requires only one message: the request, which carries any necessary data, to perform the update.

procedure call (RPC)[5] would be too high to use for transposition tables.[6] Recent work on *optimistic active messages* promises to obtain the benefits of general RPC's at a cost comparable to an active message [HJK*94]. Optimistic active messages require that a message handler be abortable, which introduces additional complexity in the design and implementation of message handlers.[7] The

[5]For an example of remote procedure call (RPC) see [BN84].

[6]For example, on a DECstation using an optimized RPC with an ATM network, the message-passing overhead is about 170 microseconds [TL93].

[7]Recall also the discussion in Section 4.6 on how aborting entire subcomputations is accomplished in StarTech.

StarTech scheme is simpler and faster, because it does not require abortable handlers or locks.

6.4 Improving the Transposition Table Effectiveness

The transposition table improves performance by storing 'exact' results that can be used to avoid a search of an entire subtree and by storing 'partial' results that can be used to improve move-ordering. The 'exact' results are encountered when, due to transpositions, the exact same position is searched more than once during a tree-search. The result from the first search is stored in the transposition table. Then, at the start of the second search, the transposition table is interrogated, the result is found, and the entire second search is avoided. This method of avoiding reconvergant searches is *not* the most important use of the transposition table, however.

The most important use of the transposition table is to store 'partial' results which can be used to improve move-ordering. As an example of using partial information to improve move-ordering, consider a search of some position p to depth k. The best move for the depth k search is very likely to be same move as would be gotten from a depth $k-1$ search. If we have already searched position p to depth $k-1$ and stored the move in the transposition table, then we can use the result from the shallower search to improve the move-ordering for the deeper search. In this case, the transposition table contains an 'exact' result for depth $k-1$, but only contains a 'partial' result for a depth k search: a hint as to the best move. To improve the effectiveness of the transposition table, we tried to improve both the effectiveness of 'exact' results and 'partial' results.

StarTech uses the standard transposition table hash mechanism developed by Zobrist [Zob70] and used by most modern chess programs. Using Zobrist's method, we think of the entire state of a chess position p as being a collection of properties, such as whether there is a particular piece on a particular square. Chess positions include properties such as "there is a white pawn on a1" and "Black has queenside castling rights". We define the hash key of position p as

$$h(p) = \bigoplus_{P(p)} h_P,$$

where h_P is a wide integer (64 bits in StarTech) associated with property P (the hash code of P), and $\bigoplus$ is the bitwise exclusive-or operator. That is, $h(p)$ is the bitwise-xor of the hash codes of all the properties that are true of chess position p. Part of the hash key is used to index the array of transposition table entries, and part of the hash key is stored in the hash table to help detect false hits.

The choice of values for the hash codes h_P affects the performance of the chess program. If the array index part of two hash keys are the same for two different positions, then the two positions collide in the transposition table. We want positions that are searched near each other in time to have different hash keys. The standard method is to choose random values for the hash codes. The Hitech program, from which StarTech was developed, uses better values for the hash codes. The hash keys are chosen so that the Hamming distance between any two keys is large. One can think of choosing points that are mutually far apart in a 64-dimensional hypercube. The idea is that it takes quite a few differences between two positions in order to get the hash keys to be the same. According to Berliner, these better values produce a measurable performance improvement [Ber91] for the Hitech program. The StarTech program uses the hash keys from Hitech.

In any cache, the replacement policy affects the cache hit rate. Given two values that both want to be stored in the same entry of a direct mapped cache, such as the StarTech transposition table, one must choose which value to keep. One may prefer to keep a value that corresponds to a deeper search, since that value contains knowledge that would be expensive to recompute. One may also prefer to keep a value that is more recently used because that value is more likely to be useful in the future than a less recently used value.

I tuned the transposition table replacement policy to try to ensure that more important data is kept rather than less important data. I performed only crude tuning. StarTech's replacement policy is to keep deep positions rather than shallow positions, and if two positions are of the same depth, StarTech keeps the more recently used position. StarTech's policy is less sophisticated than Hitech's policy which, roughly, is to keep the most recently used value unless it is the result of a very deep search. In the future, I need to spend more time to carefully tune the replacement policy.

For my experiments, the transposition table is large enough that the replacement policy is not very important. StarTech searches about 1200 positions per second per processor, and has 2^{21} transposition table entries per processor, and so it takes about 800 seconds to search enough positions to fill half the transposition table. The longest search in my set of experiments is less than 400 seconds for 128-processor machine.

Recursive iterative deepening is related to the iterative deepening used by most chess programs (see for example [SA77, Gil78]). Normal iterative deepening involves searching from the root to depth 1, then to depth 2, then to depth 3, and so forth. Normal iterative deepening has among its advantages the fact when a search of position p to depth k is started, the transposition table typically has the result for a search of position p of depth $k - 1$. Since part of the result stored in the transposition table is the best move, the program gets a good hint about what the best move is. Sometimes, the table does not have the result of the search to depth $k - 1$ for position p, in which case the program proceeds to do the search of depth k with almost no knowledge about the tree. Recursive iterative deepening rectifies this situation. If the value of the search to depth $k - 1$ is not in the tree, then we search (recursively) to depth $k - 1$ before searching to depth k.

Iterative deepening was first described by D. Slate and L. Atkin [SA77]. T. Truscott states that he implemented recursive iterative deepening in a checkers program [Tru92] that had no move-ordering heuristics except for the transposition table. Truscott says that he never attempted to use recursive iterative deepening in his Duchess [Tru81] chess program. H. Berliner states [Ber93] that in the middle to late 1980's the Hitech team identified something they called a *search catastrophe*. A search catastrophe usually happened on the last iteration of a search when the move that had been preferred up to that point suddenly collapsed. In this situation, the transposition table only contained enough information about the other moves at the root to find some refutation. Such information could be quite far away from anything resembling the best line of play. So poor Hitech would spend much time on deep and uninformed searches of bad moves. To combat the search catastrophe Murray Campbell and Gordon Goetsch wrote a recursive iterative deepening program. Berliner could not recall whether the recursive iterative deepening was made a permanent part of Hitech.

Figure 6-5 shows the effect of recursive iterative deepening on our serial program. In the serial program, recursive iterative deepening slows things down , for positions which are solved quickly. The value of improving the move ordering for such a short program is small, and the transposition table is likely to hold the right answer, so little can be gained. For the slower positions, recursive iterative deepening helps more, providing more than a factor of three speedup for the slowest position. Recursive iterative deepening helps the serial program precisely when the transposition table suffers from collisions.

Sometimes a parallel program can search the same tree multiple times in parallel. A parallel chess program must cope with reconvergant game trees differently than a serial chess program does. The transposition table helps improve the efficiency of the search, because when a position has been searched before, the transposition table remembers the value. Thus, in a serial program, every transposed position can be caught by the transposition table, and transposed positions are only

Position	Without RID time (seconds)	With RID time (seconds)	Speedup
22	0.04	0.05	0.79
21	0.82	0.85	0.97
23	1.07	1.08	0.99
16	1.97	2.11	0.93
19	32.63	55.34	0.59
13	37.68	36.27	1.04
11	44.35	39.04	1.14
20	74.45	73.50	1.01
9	76.35	69.03	1.11
12	88.00	83.90	1.05
18	104.15	80.87	1.29
5	128.02	111.80	1.15
10	226.30	189.75	1.19
17	247.26	237.41	1.04
8	341.99	243.97	1.40
14	681.66	512.67	1.33
25	774.18	872.94	0.89
6	1476.79	895.57	1.65
3	1600.24	869.57	1.84
15	2088.25	1012.29	2.06
24	2220.37	2037.59	1.09
2	6207.41	2850.86	2.18
7	14224.06	3616.37	3.93
1	14553.25	7370.59	1.97
4	16059.45	5182.52	3.10
Top 20	8026.20	5388.02	1.49
All	61290.74	26445.96	2.32

Figure 6-5: The effect of recursive iterative deepening (RID) on the serial program. Each of Kaufman's problems was to solution on our serial implementation. The time, in seconds, to solution is shown with and without RID for each problem. The problems are sorted according to the time taken without recursive iterative deepening.

missed due to the transposition table being too full. In a parallel implementation, it is possible for two different processors to reach the same position at about the same time, for both processors to miss on the transposition table, and then for both processors to perform the same work redundantly. Then other processors steal work from those two processors, and many processor-seconds can be wasted.

I designed a way to catch these parallel transpositions. The idea is that when one of the processors does a transposition-table lookup on a position, it records the fact that the position is being searched in the transposition table, and then when a second processor does a transposition lookup, it notices that the search has already started, so it just waits until the search completes and then uses the value. In such a scheme, the second processor sits idle while waiting for the transposition-lookup. It is better to sit idle, however, than to start redundant work, some of which may be posted and subsequently stolen by other processors, causing them to do redundant work too.

To implement the waiting, I used a technique developed for I-structures [ANP89]. The transposition table lookup is performed as a pair of messages: A "lookup" message is sent from the frame to the transposition table, and a "data" message is used to send the reply back to the processor from the transposition table. We want to have the second processor wait until the first processor finishes, indicated by sending an "update" message. All we do to implement the wait is hold onto the "data" message until the "update" arrives. The second processor just sits in a polling loop waiting for its data to arrive, thus implementing the wait.

Such a scheme must be careful to avoid deadlock. If a lookup on some position p to depth 3 were to be held up by a lookup on position p of depth 7, for example, we might deadlock, because the depth 7 search might need the value from the depth 3 search to complete. We adopt the following rule:

> If there is a search of depth k in progress, then defer the lookup of any search to depths j if and only if $j \geq k$.

Thus there could still be multiple searches of the same position at the same time, but once a search of a position to a given depth starts, only shallower searches of that position may be started concurrently. When combined with recursive iterative deepening, this rule means that only one particular search of a particular position actively spawns work at any time. The deeper search ends up waiting for the shallower search to complete before continuing with the deeper search. I found that if the program defers a search of the same position with an unrelated α-β window, then the performance becomes much worse, because the unrelated windows do not actually depend on each other.

I measured effect of recursive iterative deepening and deferred reads on the performance of StarTech. Figures 6-6 through 6-8 shows the performance of StarTech on each of Kaufman's test positions with and without deferred-reads and with and without recursive iterative deepening, for various machine sizes. As for the serial program, recursive iterative deepening dramatically improves the performance of the program. The time to solve the top 20 problems is reduced to 246.9 seconds using both mechanisms, which according to Kaufman's formula (Equation 6.1) places StarTech's estimated rating at 2451 USCF.

Surprisingly, the deferred-read mechanism increases the variability of the performance by $\pm 20\%$ from one execution to the next. The average performance with the deferred-read mechanism is slightly better than without, but the range of measured performance for the same size machine is much wider. I determined that the variation is actually in the amount of work, rather than, for example, because of increased waiting time due to the deferred read mechanism. I do not yet understand why the variation is so much higher with deferred reads. It may pay to study how the deferred read function is affected by the type of chess position being searched or by the use of a k-way associative transposition table instead of a direct-mapped transposition table.

The deferred-read mechanism increases the total amount of time waiting for transposition table lookups from 5% of all the processor cycles to 15% of all the processor cycles. This waiting-time increase illustrates a tradeoff in my scheduling strategy. It would be nice to be able to execute work from some other part of the tree while waiting for the work to complete. But I wanted to get all of the advantage of the normal C-language call stack for the case where work is not being stolen, so I did not use a general heap of activation-frames representation to represent the local computation. If I used a general heap of activation frames [NA89, Hal94], or perhaps a threads package [Bir89], the time wasted waiting for hash lookups to complete could be used for something else. Perhaps

RID		no	no	yes	yes		no	no	yes	yes
DR		no	yes	no	yes		no	yes	no	yes
C	Pos 4	10.01	5.78	10.89	5.48	Pos 5	3.11	3.21	4.09	3.10
W		68003.1	123513.4	25958.0	7540.4		1439.9	1298.0	1347.9	1422.0
T		141.1	239.9	62.7	24.7		13.7	10.9	15.2	11.2
C	Pos 1	15.46	9.12	41.28	15.44	Pos 18	4.13	2.41	3.92	2.81
W		59340.5	78419.6	59484.3	38510.3		1985.1	1026.6	1078.4	540.0
T		132.2	159.0	115.2	88.4		12.6	10.9	12.7	9.9
C	Pos 7	14.20	9.13	16.89	17.02	Pos 12	3.50	2.62	3.48	4.15
W		52277.3	64950.4	24192.5	65713.8		1238.5	2424.2	952.5	1602.4
T		114.7	133.7	67.3	139.8		12.9	13.2	12.5	12.2
C	Pos 2	19.35	4.28	11.00	8.34	Pos 9	3.23	2.20	2.80	3.08
W		33258.6	10540.2	21506.1	22997.0		1627.0	625.4	1619.4	1114.2
T		74.4	29.4	54.1	53.7		11.7	10.5	11.3	9.9
C	Pos 24	9.68	8.14	13.12	12.00	Pos 20	3.74	3.67	4.23	3.28
W		20234.7	22651.0	14562.3	21351.6		632.9	1277.3	759.2	1003.7
T		51.2	52.9	49.0	54.0		16.1	11.6	14.9	11.3
C	Pos 15	4.95	6.57	5.31	3.89	Pos 11	2.83	2.61	2.57	3.27
W		10396.5	17383.4	6352.1	7148.8		511.3	492.4	213.8	446.8
T		29.9	43.8	22.5	22.4		11.1	10.1	10.9	9.6
C	Pos 3	4.65	4.08	9.43	6.25	Pos 13	2.64	1.84	2.25	3.31
W		11539.5	12110.2	10335.7	8626.1		700.3	543.0	439.8	762.5
T		34.7	32.5	42.5	28.1		9.3	8.9	9.1	9.3
C	Pos 6	19.07	2.35	10.91	3.25	Pos 19	2.73	2.41	3.68	3.62
W		20409.9	291.0	6347.0	1609.7		371.1	266.4	497.2	526.1
T		60.9	9.5	30.3	10.6		10.5	9.4	11.4	11.2
C	Pos 25	8.50	8.12	12.02	8.40	Pos 16	1.42	1.29	1.39	1.34
W		5572.3	4770.7	4843.5	4231.0		31.5	22.5	63.1	41.2
T		24.1	22.8	27.2	22.1		6.5	8.4	8.0	7.0
C	Pos 14	5.64	4.29	5.02	4.65	Pos 23	1.07	0.69	0.99	0.95
W		5967.4	7566.4	3138.7	5918.8		33.9	19.9	11.7	19.4
T		23.3	24.2	17.5	21.5		7.1	5.9	6.0	5.9
C	Pos 8	3.97	2.92	4.89	4.18	Pos 21	0.96	1.22	1.06	0.87
W		3485.7	3614.0	2064.1	3391.2		11.0	14.7	13.8	8.6
T		16.7	16.1	15.4	16.0		5.5	6.0	5.9	6.2
C	Pos 17	3.50	2.49	3.94	4.95	Pos 22	0.49	0.49	0.41	0.41
W		2672.1	1058.7	1642.1	5328.3		0.8	1.1	1.1	1.0
T		13.3	10.9	11.5	21.6		4.1	4.1	4.1	4.2
C	Pos 10	3.55	2.66	3.60	2.98	Totals	148.82	91.94	175.55	124.05
W		1751.9	1762.9	1264.6	582.7		301740.9	354880.5	187424.2	199854.8
T		13.8	13.3	11.3	9.4		837.6	884.6	637.3	610.9

Figure 6-6: The effect of deferred reads and recursive iterative deepening on the parallel program, running on 512 processors. Each of Kaufman's problems was run to solution. The time is shown in seconds for the combinations consisting of with and without deferred-read and with and without recursive iterative deepening. The problems are sorted according to the serial time without recursive iterative deepening.

RID		no	no	yes	yes		no	no	yes	yes
DR		no	yes	no	yes		no	yes	no	yes
C	Pos 4	7.08	3.42	8.55	4.23	Pos 5	1.94	1.53	2.43	1.96
W		27215.1	12130.9	15515.0	11462.5		491.4	616.0	728.6	679.5
T		223.6	106.5	135.4	100.3		14.6	13.0	16.9	15.0
C	Pos 1	8.91	5.98	12.20	7.10	Pos 18	2.06	1.65	2.28	2.24
W		28581.2	39552.5	18548.4	20565.4		674.3	615.3	591.6	437.6
T		240.7	322.0	167.4	173.2		14.8	13.9	14.5	12.6
C	Pos 7	11.39	6.76	12.30	4.59	Pos 12	2.41	1.32	1.75	1.04
W		28379.8	11547.9	13219.3	5386.9		683.9	1124.1	407.0	216.6
T		235.9	102.0	122.0	54.0		16.6	16.5	12.6	10.0
C	Pos 2	5.13	2.84	6.43	4.45	Pos 9	1.33	0.92	1.55	1.13
W		15733.7	21916.9	12419.4	9856.1		731.4	180.9	644.0	292.3
T		133.6	181.4	108.5	88.5		15.6	9.4	13.4	10.7
C	Pos 24	6.47	5.33	7.83	6.07	Pos 20	3.15	1.94	3.22	1.53
W		13804.6	11255.7	7030.7	6768.0		440.8	738.6	401.4	251.1
T		119.7	99.6	71.1	66.2		18.0	14.5	15.8	10.4
C	Pos 15	2.89	1.95	3.06	2.67	Pos 11	1.44	1.44	1.86	1.22
W		5661.2	7725.2	3641.0	3470.4		299.9	250.1	167.2	181.2
T		54.8	70.8	39.4	37.8		11.4	9.9	11.3	9.8
C	Pos 3	3.78	1.50	5.38	4.08	Pos 13	1.44	1.36	1.84	1.41
W		10376.4	6323.4	6415.9	5164.1		355.4	364.7	275.6	287.4
T		95.9	59.2	69.5	51.8		11.0	11.1	10.8	10.2
C	Pos 6	7.74	1.14	7.57	1.58	Pos 19	1.96	2.17	2.67	2.39
W		10734.9	159.3	4046.5	818.2		276.5	507.4	300.6	258.4
T		107.9	9.6	48.6	14.2		12.6	13.8	13.3	12.1
C	Pos 25	5.91	5.69	8.78	7.18	Pos 16	0.85	0.88	0.88	0.78
W		3480.7	3690.4	3178.7	2956.6		12.9	26.8	14.8	16.0
T		39.7	41.5	40.8	38.4		7.4	7.6	6.9	7.0
C	Pos 14	3.28	2.19	3.03	2.66	Pos 23	0.60	0.45	0.43	0.42
W		3242.7	3858.4	1878.9	4168.6		12.6	7.4	5.3	4.8
T		38.3	40.6	26.0	44.0		7.3	6.2	6.0	5.8
C	Pos 8	1.83	1.78	3.01	2.26	Pos 21	0.58	0.69	0.53	0.57
W		1430.5	2821.1	1119.0	1577.1		7.4	15.6	4.9	5.8
T		20.6	32.0	19.4	22.0		6.6	6.6	6.1	6.0
C	Pos 17	2.46	1.25	2.53	3.82	Pos 22	0.21	0.18	0.20	0.18
W		1365.3	424.3	864.0	3013.7		0.4	0.3	0.5	0.4
T		19.6	12.6	15.1	36.1		4.1	4.1	4.1	4.1
C	Pos 10	2.23	1.66	2.09	1.39	Totals	84.85	54.36	100.34	65.54
W		1416.0	654.1	650.8	400.6		153993.2	125853.0	91418.3	77838.8
T		22.1	15.1	14.0	11.6		1470.4	1204.3	995.0	840.2

Figure 6-7: The effect of deferred reads and recursive iterative deepening on the parallel program, running on 128 processors. Each of Kaufman's problems was run to solution. The time is shown in seconds for the combinations consisting of with and without deferred-read and with and without recursive iterative deepening. The problems are sorted according to the serial time without recursive iterative deepening.

RID		no	no	yes	yes		no	no	yes	yes
DR		no	yes	no	yes		no	yes	no	yes
C	Pos 4	5.60	2.34	7.23	3.26	Pos 5	1.65	1.23	1.82	1.42
W		27754.8	2611.6	12988.2	8324.5		421.1	402.5	410.0	480.4
T		872.6	92.4	414.3	273.0		24.0	20.9	23.0	23.4
C	Pos 1	9.29	5.41	13.45	9.07	Pos 18	1.65	1.38	2.25	2.93
W		21298.7	23526.0	17894.3	24677.5		496.2	364.4	418.5	695.0
T		681.4	756.0	575.7	794.2		23.9	20.2	22.4	32.8
C	Pos 7	10.66	7.23	14.13	9.78	Pos 12	1.60	0.69	1.84	0.88
W		19645.6	22451.7	12446.0	42384.0		420.9	125.9	255.7	96.3
T		622.0	718.1	401.7	1350.7		23.3	11.7	17.7	11.2
C	Pos 2	6.99	2.27	6.85	5.10	Pos 9	1.25	1.08	1.15	0.83
W		14350.7	6486.1	9577.7	7878.2		425.1	271.9	280.3	120.8
T		459.4	215.2	311.0	261.6		21.8	17.2	16.9	11.9
C	Pos 24	5.66	4.26	8.79	6.44	Pos 20	2.84	1.84	3.01	1.69
W		9140.7	7759.3	7035.6	7685.5		286.5	713.7	310.9	340.4
T		296.3	256.5	234.7	257.7		21.7	31.8	21.3	19.0
C	Pos 15	2.73	1.90	2.59	1.95	Pos 11	1.57	1.03	1.61	0.90
W		4250.6	5543.1	2955.0	2372.0		175.4	182.8	129.2	117.6
T		143.5	186.7	102.6	84.7		14.6	13.5	13.5	11.4
C	Pos 3	3.11	1.27	5.03	3.09	Pos 13	1.32	0.96	1.66	1.19
W		6501.0	3533.7	4630.3	3657.7		199.9	162.6	173.9	194.9
T		216.3	122.5	162.7	127.3		14.1	13.3	13.8	13.9
C	Pos 6	11.21	1.31	6.81	1.36	Pos 19	1.77	1.97	2.34	2.19
W		7019.0	1998.3	2858.5	596.4		130.0	384.8	193.0	197.2
T		232.5	71.2	102.7	26.8		13.3	21.5	15.9	15.5
C	Pos 25	5.33	4.74	7.46	7.03	Pos 16	0.62	0.83	0.69	0.76
W		2878.8	2286.7	2240.8	2518.6		9.9	13.7	11.7	12.9
T		100.3	82.4	82.8	92.9		6.9	8.4	7.3	9.1
C	Pos 14	2.05	2.16	2.83	2.44	Pos 23	0.54	0.22	0.40	0.35
W		1767.0	2863.0	1512.8	2758.7		12.0	3.5	4.4	4.5
T		64.6	101.4	58.1	99.0		7.0	5.9	6.2	6.0
C	Pos 8	1.99	1.26	2.83	1.40	Pos 21	0.36	0.42	0.41	0.40
W		1144.3	1597.9	818.2	1097.0		3.9	5.8	4.5	4.4
T		45.5	59.9	35.6	43.1		5.8	6.0	6.8	6.6
C	Pos 17	1.90	1.09	2.41	3.02	Pos 22	0.17	0.11	0.17	0.12
W		648.7	287.2	726.1	1542.9		0.4	0.4	0.5	0.4
T		28.9	18.1	31.3	58.1		4.3	4.2	4.3	4.3
C	Pos 10	1.90	1.10	1.61	1.31	Totals	81.85	47.03	97.75	67.62
W		1005.0	202.3	443.3	487.1		118981.2	83576.7	77875.8	107757.7
T		41.7	14.9	22.3	23.5		3943.9	2855.0	2682.5	3634.2

Figure 6-8: The effect of deferred reads and recursive iterative deepening on the parallel program, running on 32 processors. Each of Kaufman's problems was run to solution. The time is shown in seconds for the combinations consisting of with and without deferred-read and with and without recursive iterative deepening. The problems are sorted according to the serial time without recursive iterative deepening.

even having as few as two main threads of control would do the job. This problem is related to the strategy of waiting for children (discussed in Section 5.8). In both cases, the processor waits instead of doing some useful work. It makes sense to investigate some of these alternative scheduling strategies in the future.

6.5 Efficiency Heuristics for Jamboree Search

Another approach to improving the performance of StarTech is to reduce the total work by serializing the search. I found a few heuristics that can improve the work efficiency of the Jamboree chess algorithm on real chess positions. This improvement in efficiency often comes at the expense of an increased critical path length. I did find one heuristic that actually improves the performance without increasing the critical path significantly, however.

I first set out to identify what work is wasted. There are two cases where the Jamboree algorithm does work that is not necessary:

failed work is work done to test a position when the test fails, and the position must be searched for value. Some of the failed work is a cost introduced by the serial Scout algorithm, since serial Scout also performs a research. Some additional failed work is incurred, because in the serial search the test is possibly performed with a tighter bound than is available during the parallel search.

cutoff work is work that is done on a child of a position that would not have been expanded in a serial execution because an earlier child would have failed high.

In addition, for a subsearch that is neither a failed test nor cutoff work, we define the *failed work* of the subsearch to be the sum of the failed work of its children, and the *cutoff work* of the subsearch to be the sum of the cutoff work of its children. I arranged for StarTech to computed the amount of failed work and cutoff work. I found that most of the inefficiency of Jamboree search is cutoff work. Depending on the position, 10% to 30% of all the work is cutoff work, while less than 2% of the work is failed work.

I furthermore examined the conditions under which work is stolen. I categorized the conditions by the Knuth-Moore position type [KM75]. Rephrasing the Knuth-moore position types in terms of Jamboree search:

- Type 1 positions are the positions searched by Jamboree search with a non-empty window.
- Type 2 positions are the positions searched with an empty window that have an even number of ancestors between them and the nearest Type 1 position. Thus, the tested children of a Type 1 position are Type 2 positions, as are the grandchildren of any Type 2 positions.
- Type 3 positions are the positions searched with an empty window that have an odd number of ancestors between them and the nearest Type 1 position. Thus the children of a Type 2 position are Type 3 positions, and the children of Type 3 positions are Type 2 positions.

I also categorized the extra work by one of three conditions when a position is searched to depth k. This categorization uses two values:

s_{k-1}**:** The value of the position, searched to depth $k-1$. The value s_{k-1} is either obtained from the transposition table or by recursive iterative deepening.

c_{k-1}**:** The value of the first child of the position, searched to depth $k-1$. The first child is identified using the move-ordering heuristics. In particular, the first child is the best move for the search of the position to depth $k-1$.

The categorization of the extra work is as follows:

Better than α: The case when $c_{k-1} > \alpha$. That is, the first child is a reasonable move compared to α.

Dropped below α: The case when $s_{k-1} > \alpha$ and $c_{k-1} \leq \alpha$. That is, on the previous ply of search, we thought that the first move was good, but now that we have searched deeper, we do not like the first child.

Stayed below α: The case when $s_{k-1} \leq \alpha$ and $c_{k-1} \leq \alpha$. That is, we never did think that this was a good move, and we still don't think so.

I found that depending on the position, 50% to 90% of the failed work is on Type 2 positions that dropped below α, while 10% to 40% of the failed work is on Type 3 positions that dropped below α.

Using that data, I decided to try serializing the search for Type 2 and Type 3 positions that drop below α. This approach reduced the work by as much as 50%, which was even more than my measurements indicated that it might. The critical path was increased, however, so that the average available parallelism dropped below 100. On small machines the critical path was not a problem, but for big machines the serialization hurt the performance of the program.

I tried a finer strategy for serializing the search. My idea was to not completely serialize the positions that were causing failed work to appear, but simply to serialize the position a little bit. I tried a strategy of searching exactly one additional child serially , for positions of Type 2 that drop below α, before searching the rest of the children in parallel. This strategy worked out well, decreasing the total work by 10% to 15% while only increasing the critical path slightly, so that the average available parallelism was still over 500.

One recent enhancement to the Zugzwang program [FMM91] is to explicitly compute the number of critical children of a position, and when searching a position with exactly one critical child, and several promising moves, Zugzwang searches all the promising moves sequentially before starting the parallel search of the other children. Since the Zugzwang literature does not analyze critical path lengths, it is difficult to determine how Zugzwang's serialization scales with the machine size, however.

In summary, by measuring the critical path and total work, I was able to improve the performance of the StarTech program over a wide variety of machine sizes. If I had only studied the runtime on small machines, I would have been mislead into overserializing the program. By measuring the critical path length, I was able to predict the performance on a big machine. I then verified that the performance of the tuned code matched the prediction when run on a big machine.

6.6 How Time is Spent in StarTech

Now that we have a good understanding of the Jamboree search algorithm and the StarTech scheduler, it is worthwhile to examine how time is spent by the StarTech program. Examining a timing profile can provide important clues for how to improve the program in the future.

Figure 6-9 shows how the processor cycles are spent by StarTech on a typical chess position that ran for about 100 seconds on a 512-processor machine. The program used the deferred-read mechanism and recursive iterative deepening, although it did not use the Jamboree serialization heuristic of Section 6.5.[8] The biggest chunk of time is devoted to the chess-work, which further broken down in Figure 6-10.

[8]The Jamboree serialization heuristics do not substantially change the ratios of how processor cycles are spent.

68.8%	of the cycles is 'chess-work' done by the parallel algorithm. Of those cycles, 21.4% can be accounted for by the time that our best serial implementation consumes.
14.4%	of the cycles are spent by processors waiting for global transposition table reads to complete.
6.6%	of the cycles are spent by idle processors waiting for the global throttle to give them permission to steal work again.
3.6%	of the cycles are spent by idle processors looking for work to do.
3.2%	of the cycles are spent waiting for a child to complete, to determine if more work needs to be done at a position.
2.2%	of the cycles are spent by busy processors servicing a transposition table lookup.
0.6%	of the cycles are spent by processors that have work to do responding to a request for work.
0.5%	of the cycles are spent by a child waiting for an 'abort' message from its parent, after sending the result to the parent.

Figure 6-9: How processor cycles are spent by 512 processor StarTech running a typical problem from Kaufman's problem set, using the deferred read strategy and recursive iterative deepening.

37.7%	of all the cycles are spent on control flow for the Jamboree algorithm.
15.8%	of all the cycles are spent moving the pieces on the board.
8.3%	of all the cycles are spent on static evaluation.
3.3%	of all the cycles are spent on move generation.
2.0%	of all the cycles are spent sorting the moves.
1.6%	of all the cycles are spent checking for repeated positions.
0.2%	of all the cycles are spent checking for illegal moves.
68.8%	of all the cycles are spent on 'chess work'.

Figure 6-10: How the time is spent on 'chess work' for StarTech running on 512 processors on a typical problem.

More than a third of all the processor cycles, and more than half the cycles spent by on 'chess work' are spent by the code that implements the control flow of the Jamboree algorithm. In the serial program, the control-flow of the α-β search algorithm consumes about a quarter of all the processor cycles. The biggest potential improvement is to improve the code that executes the Jamboree search, although I have not been able to find any obvious improvements to the code.

Using true multithreading could potentially get 14.4% of the cycles back from waiting on transposition table reads, 3.2% of the cycles from the time waiting on children, and 0.5% of the cycles spent waiting on parents. To save those 18.1% of the cycles would require implementing more code in the scheduler to handle context switching and frame allocation. These improvements are worth investigating.

It may be possible to tune the global throttle to get back a few percent of the runtime. In this example, the StarTech program spends 6.6% of the cycles with processors being stalled by the global throttle. Since the processors then spend 3.6% of the time actually sending requests for work, and waiting for the reply, the throttle is not causing too much trouble. Only 0.6% of the cycles are spent by processors actually responding to requests for work. If we open the global throttle a little bit, it will increase that 0.6% which impacts on the efficiency with which work actually is executed. On the other hand, we may be able to get back some of the time spent waiting on the throttle. In addition, if we were using true multithreading, we could potentially hide some of the time spent waiting on the global throttle and waiting on the requests for work. This improvement is probably also worth investigating.

It has been widely argued that using the Hitech static evaluator is a bad match for an all-software implementation. Since Hitech uses special purpose hardware, the Hitech static evaluator expects to run in constant time regardless of how sophisticated the static evaluation function becomes. So the Hitech static evaluation function is designed to be as sophisticated as possible given the constraints of the Hitech hardware. In StarTech only the 15.8% of the cycles spent moving pieces on the board and the 8.3% of the cycles spent on static evaluation are attributable to the Hitech emulation. Perhaps a static evaluator designed for a software-only system would be better than Hitech's static evaluator, but it probably can't be much faster.

The code for move generation and checking illegal moves, which takes a total of 3.5% of the cycles, was optimized in assembly language by Ryan Rifkin under the direction of Mark Bromley of Thinking Machines Corporation. Before Ryan worked on that code, the move generation and illegal move checking accounted for about 9% of all the cycles.

To check for illegal positions correctly and efficiently required some careful design. Recall that in chess, if a position is repeated three times, then either side can call the position a draw. Most computer chess programs actually do claim a draw as soon as a position is repeated thrice. Most chess programs, however, treat illegal positions improperly while performing the search. There are three cases when looking at a position in the search tree:

- If the position has appeared twice in the game (i.e., before the root of the game-tree search), then the position should be evaluated as a draw. Most programs do this correctly.
- If the position has not appeared at all in the game, then the position should be evaluated as a draw if and only if the position appears as an ancestor in the search tree, because the only rational reason to return to the same position is to try to force a draw. Most programs do this correctly as well.
- If the position has appeared once in the game (before the root), then the rule should be the same as if the position has appeared zero times in the game, because the program should not assume that the other player is playing rationally. Most programs treat this case as a draw,

and it is possible for a clever opponent to trick the computer into turning a draw into a loss for the computer.

The summarized rule is thus:

StarTech Repeated Position Rule:

> Evaluate a position as a draw if it appears once as an ancestor in the search tree, or if it appears twice in the actual game.

As far as I know, this rule is unique to StarTech. Most programs use a simpler rule:

(Slightly) Wrong Repeated Position Rule:

> Evaluate a position as a draw if it appears once as an ancestor in the search tree, or if it appears once in the actual game.

Here is how StarTech implements its repeated move tests. The entire game is broadcast to all the processors, so that the processors are able to examine the repeated move list. StarTech uses the hash keys from the transposition to test if two positions are the same quickly. When a position is stolen, the sequence of positions between the root and the stolen position is sent through the data network to the stealing processor. This sequence of positions can be thought of as the part of the board-state.

We can improve the performance of the repeated move check by looking towards the root from a position only until an irreversible move is detected. An *irreversible move* is a move (such as a capture, a pawn move, or a move that results in the loss of castling rights) that guarantees that no previous position can be repeated. In fact, when sending the sequence of previous positions from one processor to another, stealing processor, StarTech only sends the positions as far back as the most recent irreversible move. Since in quiescent search, which accounts for more than half of all the positions searched, almost all moves are irreversible, the StarTech program spends very little time manipulating repeated position sequences.

One of the biggest open questions for tuning parallel chess programs is the impact of additional search heuristics on the critical path and total work. In StarTech we only did a simple search to a given depth and then performed quiescence search, trying out all the captures. Most state-of-the-art chess programs employ search extensions and forward pruning to improve the quality of their tree search. We have yet to see whether those sophisticated serial search strategies can be effectively parallelized.

In this chapter, we have studies ways to improve the performance of StarTech. These, along with the strategies studied in Chapter 3, should be useful for improving the performance of any parallel program.

Chapter 7

Conclusions

This dissertation argues that fast, predictable, global synchronization can solve many of the system-problems of parallel computing. The dissertation first described hardware that uses global synchronization, then described a dynamic MIMD-style chess program which profitably exploited global synchronization, and finished with a study of how to use global synchronization to obtain good performance on bulk data transfers. Here I review the contributions of my work, discuss related work, and speculate about future application of the mechanisms that I developed.

Architectural Support for Global Synchronization

The computer architecture work of my thesis consists of the design of the Connection Machine CM-5 supercomputer, which was originally intended for running data-parallel programs. Global synchronization is used throughout the CM-5 to help solve low-level problems, such as clock distribution and diagnostics support, and higher level problems of operating system support and running data-parallel programs.

The three networks of the CM-5 (the control network, the data network, and the diagnostics network) fit into an framework called the synchronized MIMD architecture. A method of executing data-parallel programs on a synchronized MIMD computer was presented, along with a sketch of how to optimize such programs. The details of a specific synchronized MIMD computer, the CM-5, were presented to show what kinds of problems and solutions can be addressed by exploiting global synchronization. The CM-5 and its networks can be timeshared, and can be spaceshared by dividing the machine into independent partitions. The networks are clocked using a robust, globally synchronous clocking system and communicate using low-swing electrical signals.

The processor-network interface allows fast user-level access to the networks without compromising interprocess security. The network interface uses address translation to guarantee that the user cannot send a message outside his own partition. The system uses a globally synchronous operating system to guarantee that when a message arrives at a processor, the message belongs to the process that is currently running.

The data network provides fast pairwise communication among the processors of the machine. The data network is implemented as a 4-ary fat-tree that uses distributed switches. The data network is actually split into two independent networks to help the user implement commonly used protocols such as *remote-fetch*. The data network provides an *all-fall-down* primitive to empty the network for context-switching.

The control network provides global barriers, broadcast, segmented scans, and segmented reduction operations. The data network and the control network also cooperate using "Kirchhoff counting" to compute when the data network has delivered all its messages. The control network provides split-phase global synchronization.

The CM-5 diagnostics network extends the JTAG standard to test a system in parallel. The diagnostics network uses bit-steering to address subsets of the machine, so that they can be manipulated as a group. In order to make the diagnostics network robust and easy to fix, it is implemented with PAL chips that run synchronously at low speed.

The CM-5 design team tried to think of all the problems that show up in parallel programming and to provide mechanisms to solve those problems. It is a measure of the success of our effort that a whole collection of new problems have arisen that we never imagined. If we had not solved the problems we knew about, we would never see the new problems. The fact that, for the most part, those problems appear to be solvable for the CM-5 indicates that the mechanisms we designed are not overly specialized for data-parallel computing. The biggest problems for the CM-5 are faced by the operating systems implementors. Today's operating systems were not designed for parallel computing, and they are large and difficult to modify.

The history of SIMD machines goes back to the Illiac-IV [BBK*68], which had processors with wide datapaths. The terms 'SIMD' and 'MIMD' were coined by M. Flynn [Fly66]. The Goodyear MPP [Bat80], implemented with VLSI chips, started the trend towards using narrow processors — the MPP used 1-bit processors. The MPP and the Illiac-IV both provided a two-dimensional mesh for communications, and the MPP extended the idea to support a limited set of additional permutations. The TRAC [LT77] allowed the word width of the machine to be varied to match the problem, and it also provided a statically configurable Banyan network. Both TRAC and PASM [SKS81] were multiple-SIMD machines that were designed to allow a small collection of SIMD programs to run concurrently and communicate with each other. Thinking Machine Corporation's Connection Machine CM-1 [Hil85] was the first SIMD machine to provide a routing network, to let users send messages to arbitrary processors without worrying about how the messages get to their destinations. The CM-1 router used an adaptive routing scheme based on a 12-dimensional hypercube. The Connection Machine CM-2, also developed by Thinking Machines, started the swing back to wide processors by including, at first, 32-bit floating-point units, and then, later, 64-bit floating-point hardware. The CM-2 also provided indirect addressing to its local memory. The CM-2's router implemented combining both for *send* and *get* operations. IBM developed the GF11 [BDW85], a SIMD machine with wide processors and a permutation network capable of supporting up to 1024 user-defined permutations. MasPar developed the MP-1 and MP-2, both of which use 4-bit wide processors and an oblivious router based on a butterfly topology.

One of the earliest MIMD parallel computers was the Cosmic Cube [Sei85], which provided an oblivious routing network based on the hypercube. W. Dally [Dal87] argued that low dimensional meshes provide better performance than high dimensional hypercubes for oblivious routers. The architecture of the cosmic cube was commercialized by Intel, which persuaded by Dally's arguments, eventually switched to an oblivious two-dimensional mesh network in the IPSC2 [Int88]. The CM-5 design team was also influenced by Dally's argument that a data network should be organized into communications channels that are neither too wide nor too narrow. Wider channels increase the diameter of the machine increases, increasing latency, while narrow channels increase the time it takes for a message to cross a single channel. When designing the fat-tree based routing network [Lei85], we chose to use 4-bit wide communications channels.

Like the Cosmic Cube and its descendants, the CM-5 uses a distributed memory organization, and the processors communicate among themselves using message-passing techniques [Sei85, SAD*86]. Another way to organize the memory of a MIMD computer is as a shared-memory [LB80, GGK*83, DT90]. Some recent machines, such as Alewife [ACD*91], have tried to merge the shared-memory and distributed memory architectures.

Other machines with global synchronization include the SIMD machines (discussed above), the proposed Burroughs Flow Model Processor (FMP) [LB80], the DADO machine [SS82], and the DATIS-P machine [PS91]. Of particular note is the FMP, which although it was never built, proposed hardware for split-phase global barriers, and described a scheme to broadcast the same program to every processor which would then run it locally.

Split phase barriers were described for the FMP, and were also described by C. Polychronopoulos [Pol88] and R. Gupta [Gup89]. Arvind and R. Iannucci [AI87] present general justification for split-phase operations. B.-H. Lim [Lim91] performs a study to determine when it is better to simply wait and when it is better to try to use split-phase transactions.

The CM-5 does not include any message combining mechanisms in the data network. G. Pfister and V. Norton showed that combining can help avoid hot spots in a data network [PN85]. A. Chien found that combining does not work very well for many real programs, but that it is possible to dynamically detect hot spots [Chi86]. P.-C. Yew *et al.* explored how to use software combining to avoid hot spots [YTL87]. W. Dally analyzed the hot-spot behavior of meshes [Dal87].

The data-parallel style of programming developed incrementally over time. The programmers of the earliest SIMD machines were using a form of data-parallel programming embodied in FORTRAN. (For example see [Per87] for a discussion of Illiac IV CFD FORTRAN and other subsequent parallel FORTRAN dialects.) D. Christman [Chr83] compiled a collection of data-parallel algorithms and programming strategies. C. Lasser developed a programming language SIMPL (SIMd Programming Language) [Las85] based on those strategies. The SIMPL language eventually became StarLISP [Thi86a]. W. Hillis and G. Steele [HS86] identified the 'data parallel' style of programming and argued that it was more useful than had previously been thought. G. Steele [Ste90] provided a sophisticated rule to run asynchronous programs deterministically, in contrast to my simple rule of placing a barrier before and after every communication operation of a data-parallel program. G. Blelloch [Ble90] showed that the scan primitive is an essential ingredient to the data-parallel paradigm. Meanwhile much research on how to execute data-parallel programs on MIMD machines was being performed. L. Valiant's bulk-synchronous parallel model (BSP) [Val90] and Sabot's paralation model [Sab88] both attack the problem of how to express general parallel program using an underlying serial model of computation. F. Darema-Rogers *et al.* developed the Single Program Multiple Data style of programming [DGN*86]. D. Callahan and K. Kennedy focused [CK88] on how to compile bulk data movements written in serial FORTRAN. Similarly A. Rogers, K. Pingali [RP89] and Zima *et al.* [ZBG88] follow the strategy of generating a parallel program from a serial program augmented with data distribution primitives. Other data parallel languages include CM-FORTRAN [AKL*88], FORTRAN 8X (now FORTRAN 90) [MR90], High Performance FORTRAN (HPF) [HPF93], C* [RS87] and Dataparallel C [HQL*91].

Massively Parallel Chess

The chess work consists of a parallel game tree search and a scheduling mechanism, as well as the measurement of the performance of the program.

Chess playing ability, whether in a computer or in a human, is a multidimensional skill. Computers need a good opening book, good endgame knowledge, a good search strategy, a good evaluation function, and enough speed. The Elo chess rating system [Elo78] maps this multidimensional skill into a linear rating, and it is difficult to understand exactly how an improvement in one dimension affects the overall rating. Another way to view chess playing ability is that there is a space of chess positions, and that the box spanned by the various dimensional qualities contains those positions that the program understands how to play. My strategy was to take an existing chess program, Software-Hitech (which has a less rich collection of search extensions than the standard Hitech program, but is otherwise the same as Hitech of September 1991), and push as hard as I could in the brute-speed direction. The promise of my approach is that as one incorporates improvements in other dimensions, the "speed" dimension will not be a limiting factor. Improvements might include an improved static evaluation function (or perhaps one better suited for software than Hitech's hardware-oriented static evaluator), improved search extensions, and possibly even completely different search strategies such as B* search [Ber79] or conspiracy number search [McA88]. By "improving" the other dimensions of the chess skills, we hope to encompass many more chess positions into our "playable" space. In the future I hope to investigate how well such techniques can be parallelized.

StarTech uses *Jamboree search*, a parallelized Scout algorithm. The Scout algorithm is a variant of the short, but tricky, α-β algorithm. D. Knuth and R. Moore [KM75] provided the first published analysis of the α-β algorithm. The main advantage of the α-β algorithm is that, if the tree is ordered so that a good move is considered before a bad move is considered, the algorithm prunes away useless work and searches a tree which is much much smaller than the entire tree. Thus, the α-β algorithm rewards programs that do a good job of ordering moves in the search tree. Knuth and Moore contrasted α-β search with a *weak* version of α-β search that makes do without the α argument and achieves somewhat fewer cutoffs. They identified the *critical tree* of a game-tree as the part of the tree that must be searched by any algorithm, including α-β search, in order to identify the best move.

The simplest approach to searching a game tree in parallel is to spawn all the children of any given position and search them in parallel. This works wonderfully for game trees that are worst-ordered, because serial α-β searches the entire tree in that case, and the parallel version searches the entire tree in parallel. The problem arises when the game tree is ordered reasonably well, in which case α-β search prunes away most of the tree, but the naive parallel algorithm still searches the entire tree. The speedup achieved by this naive approach is generally limited to $O(\sqrt{P})$ on P processors [Fis84].

Thus, the challenge is to perform a parallel tree search while obtaining most of pruning achieved by the serial α-β search. Most of the approaches to effective parallel α-β search are based on J. Pearl's Scout algorithm [Pea80]. The serial Scout search algorithm, on which Jamboree is based, distinguishes between a search to determine the *value* of a position and a search to *test* to see if the value of a position is greater than a particular bound. Algorithm Scout obtains the value of the first child, and then it tries to prove that the other children are worse choices than the first child. If the proof fails for one of the children, Scout search researches the child to determine its correct value. Since testing is cheaper than valuing, Scout search is gambling that the tests succeed frequently enough to offset the cost of the researches.

Related to Scout search is aspiration search. In aspiration search, the search window is broken into several pieces which are searched independently. Because the windows are small, they often produce a result quickly. In fact, serial aspiration search runs faster than infinite window α-β search. The Greenblatt chess program used aspiration search [GEC67], and today most state-of-the-art chess programs, including StarTech, use some form of aspiration search. G. Baudet attempted to parallelize aspiration search by farming each of the search windows out to a different processor [Bau78]. Baudet found that parallel aspiration search achieves a maximum speedup of only about three to five.

My Jamboree search algorithm is a parallelization of Scout search. Jamboree search operates by searching the first child of a position, and then it tests the remaining children in parallel. Any tests that fail are sequentially researched. This approach is natural for parallel game-tree search, and variants of it have been used by several other parallel chess playing programs, including Cray

Blitz [HSN89] and Zugzwang [FMM91]. Other parallel algorithms based on Scout search include minimal tree search [ABD82], mandatory work first (MWF) [Fis84], principal variation splitting (PV-splitting) [MC82], and the Zugzwang search algorithm [FMM91].

Parallel aspiration search [Bau78] and the Karp-Zhang AND/OR tree search algorithm [KZ89, Ste92] use other strategies to parallelize α-β search.

Other approaches to game-tree search, not based on α-β search, include B* search [Ber79], Conspiracy search [McA88], and SSS* [Sto79]. These other approaches require much more memory space than the algorithms based on α-β search. Programs based on these algorithms have only recently begun to play competive chess.

Several chess programs find parallelism in the static evaluation function. The BEBE program by T. Scherzer[1] uses parallel hardware to perform static evaluation. Both Hitech [BE89] and Deep Thought [Hsu90] use parallel hardware for the static evaluation and move generation functions.

T. Marsland and M. Campbell's survey article on parallel search provides a more detailed explanation of the various parallel search algorithms [MC82]. M. Newborn provides a comprehensive history of computer chess through 1975, starting with the Turk [New75].

I have shown that improving the move-ordering decreases the critical path length and the amount of parallel work done. Thus, it makes sense to spend effort to improve the move ordering. StarTech's strategy is to

- sort the moves by the incremental improvement in the static evaluation function, and
- pull the move stored in the transposition table to the front of the list.

Several other techniques for improving move-ordering appear in the literature, including the following.

Recursive Iterative Deepening: In the late 1980's H. Berliner and his students at Carnegie-Mellon University noticed that their chess program would sometimes encounter a 'search catastrophe', in which some subsearch fails low, and then the program is making deep searches without the transposition table having any useful data. The program would spend a lot of time looking at moves that a shallow search would have discarded. They found that recursive iterative deepening was helpful for avoiding the search catastrophe, but did not come to a definitive conclusion on the value of recursive iterative deepening [Ber93]. I found that recursive iterative deepening is a definite advantage for StarTech, because move ordering is so important for a parallel chess program.

Deferred-Read of Transposition Table: I borrowed the techniques used in Arvind's I-structures [ANP89] and P. Barth's M-structures [BNA91] to implement deferred-reads of transposition table lookups. I used deferred-reads to keep different processors from working on the same thing, thus making more successful the dynamic-programming approach implicit in transposition tables.

Thompson Move Ordering: K. Thompson argues [CT82] that the moves should be initially sorted so that moves that capture the higher-valued pieces (such as queens) are earlier in the list. The program should prefer to capture the big piece with a small piece if we can. The current StarTech move ordering strategy tends to prefer to capture a big piece with a big piece, since the static evaluator often prefers to get the big pieces into the action.

[1] The BEBE program is mentioned in the parallel search survey article by T. Marsland and M. Campbell [MC82].

Killer and History Tables: R. Greenblatt introduced [GEC67] the killer table, which for each move at the root of the search tree, stores a refuting move which demonstrates that the root move is bad. The history table [MOS86] is a small set of moves that have recently caused a cutoff. If any "killer move" or "history move" appears in the move list, it is moved to the front of the move list. The idea of the killer and history table is that moves that are good in one position are likely to be good in other positions. StarTech does not currently use these mechanisms.

Further serialization: I showed that even if two children are searched serially before searching the rest in parallel, for best ordered trees, there is plenty of average available parallelism. For real chess trees, that strategy did not work very well, because the critical path length grew too much. A more limited strategy that searches two children only under certain conditions was successful, however. Zugzwang [FMM90] combines the killer and history tables with serialization. On type 2 positions of the search tree, Zugzwang serially searches the moves mentioned by the killer and history table before starting the search in parallel of the remaining children.

T. Marsland and P. Rushton characterized strongly ordered game-trees [MR74], and J. Gillogly measured the extent to which trees are best-ordered [Gil78]. Transposition tables are a memoization technique. For further information about how to use and implement memoization, see R. Bird's survey article [Bir80].

I found that I need a better collection of chess benchmarks than what I have. It is possible that some of my results have been biased by the fact that my chess problems run so quickly. With a more difficult set of benchmarks, I might get different results, although I would expect the overall results to be similar.

The StarTech program uses a work-stealing scheduler to distribute and schedule the work uncovered by the Jamboree search algorithm. In the StarTech scheduler, each busy processor posts any work that can be spawned onto another processor. Each idle processor periodically asks another random processor for posted work.

The work stealing scheduler uses a global throttle, implemented with the CM-5 control network, to keep the idle processors from swamping the busy processors with requests for work. The requests to work are organized into globally synchronous phases. During a single phase, all of the idle processors are allowed to send out one request, and they get their reply (which consists either of a "no work" message or a message containing some work). After all the idle processors get their replies, and all of the busy processors have had a chance to get some work done, they indicate, using the control network, that the next phase may begin. Thus, the work done by the computation is divided into "spawning" and "nonspawning" work. The spawning work is regulated by the global throttle, and the nonspawning work proceeds asynchronously. I found that the swamping problem gets worse as the machine size gets larger. Researchers who study schedulers on only small machines may find that their strategies do not scale to large machines.

Program Zugzwang uses a work-stealing scheduler with various deterministic strategies for determining who to ask for work. Program Zugzwang deals with the swamping problem in an *ad hoc* fashion, by disabling the requests for work at the beginning of the search. I found that sometimes in the middle of the search, the requests need to be throttled.

The DIB package [FM87] uses a work-stealing approach for distributed backtrack search. Package DIB provides several deterministic strategies for requesting work, some of which seem to suffer from the swamping problem. Apparently, one way DIB copes with the swamping problem is by forcing any processor that is denied work to wait for some fixed time before asking for work again. The DIB package is designed for robust operation in distributed systems with failures, while the StarTech scheduler is designed for fairly tightly coupled systems in which the entire machine is assumed to be robust.

The lazy task creation technique used in the Mul-T compiler of E. Mohr, D. Kranz and R. Halstead [MKH91] attempts to reduce the overhead of a work-stealing scheduler by avoiding the creation of closures and other data structures until work is actually being stolen. StarTech also reduces the overhead of stealing work by using this technique. When StarTech posts work to be stolen, it posts only a pointer to an existing board position, and a move to get to a new board position to be searched. Thus, StarTech does not incur the overhead of copying the entire state of a position until work is actually being stolen. The Mul-T compiler dramatically reduces the amount of overhead for managing parallelism, but the study [MKH91] does not compare the performance of their system to a best serial implementation (rather, it compares the performance to a serial implementation of Mul-T). Thus, it is difficult to judge the effectiveness of lazy task creation in Mul-T.

D. Nussbaum [Nus93] describes a work-stealing scheduler that tries to take advantage of locality in the parallel machine. Nussbaum reviews other locality preserving schedulers. For the CM-5, the difference between communicating with a nearby processor and communicating with a far processor is negligible, so I did not worry about trying to steal work from nearby processors.

The Parallel Continuation Machine (PCM) of M. Halbherr, Y. Zhou and C. Joerg [HZJ94, Hal94] uses a more general work migration strategy than I use in StarTech. By moving work from one processor to another, Halbherr avoids the space-time tradeoffs discussed in Section 5.8. He has not attempted to prove any time or space bounds for PCM with respect to critical path and parallel work, however. All the PCM programs that Halbherr has studied have much 'nicer' parallelism profiles than does StarTech. It is not yet clear how well PCM's design would do on a truly demanding application, such as parallel computer chess.

Several researchers have used adaptive backoff strategies to avoid swamping. The Ethernet [MB76] uses a randomized adaptive backoff strategy to avoid the swamping problem in the domain of local area networks. The adaptive backoff strategy has not been analyzed for work schedulers, however. A. Agarwal and M. Cherian study how to use adaptive backoff to improve system performance in the face of hot-spots. DIB uses an adaptive nonrandomized backoff strategy, and is tricky to tune for performance [FM87] Halbherr's PCM successfully uses an randomized backoff strategy to avoid the swamping problem, but Halbherr has only experimented with undemanding parallel programs [HZJ94, Hal94]. No analysis of adaptive backoff has been performed. The global throttle requires very little performance tuning, where adaptive backoff must be tuned to obtain good performance.

A variation of my method of removing acknowledgments from the abort/result protocol appears in distributed systems [Tv85].

The StarTech system uses active messages [vCG*92] for interprocessor communication. Although active messages restrict the handler to run quickly and without using too many resources, I was able to implement efficiently and simply the protocols I needed. Traditionally, programmers have used remote procedure calls (RPC) to implement distributed protocols [BN84], but RPC's are quite expensive [TL93]. Optimistic active messages promise to provide the efficiency of active messages with the full expressive power of RPC [HJK*94]. Optimistic active messages are slower and more complex than what is required of the StarTech active message protocols.

I developed a scheduler for the CM-5 with the dual goals of achieving linear speedup (subject to the critical path length of the computation) and staying within the memory bounds of the processing nodes of the CM-5. The Startech scheduler achieves the memory bounds, but makes fairly weak guarantees about the linear speedup. R. Blumofe and C. Leiserson [BL93], inspired by my early problems meeting these goals, identified a number of scheduling policies that achieve linear speedup and good memory bounds. Graham [Gra66, Gra69] showed that any directed acyclic dataflow graph could be scheduled to run within a factor of two of optimal on a machine with no overhead. R. Brent [Bre74] produced the formulation that states that the graphs can be scheduled to run in time no more than $C + W/P$, where C is the critical path, W is the work, and P is the number of processors.

The difficulty of measuring the critical path length for the real chess program derives from the fact that the program can abort subcomputations that have started. I found that aborting the subcomputations was valuable in practice, but one might ask if there is some way to get around the necessity of aborting subcomputations. There is some theoretical evidence that we cannot escape the necessity of aborting subcomputations. The ability to abort computations is similar to the parallel 'or' construct. The parallel 'or' construct, $\mathrm{por}(a, b)$, returns `true` if either a or b return `true`, even if one of a or b is a divergent computation. G. Plotkin showed [Plo77] that adding a parallel 'or' construct to a serial language not only changes the semantics of the language, but it changes the calculus of the language, leading to a fully abstract language, which means, roughly, that one can reason about the language exactly as mathematicians would like to. To implement a parallel 'or' construct efficiently requires the ability to abort the second subcomputation whenever the first subcomputation returns a 'true' value.

The GITA dataflow interpreter computes the critical path and work when executing an infinite-processor simulation [NFH88]. The GITA interpreter produced critical path lengths, but no guarantees were made for speedups or space bounds, partly because the dataflow programs being simulated were *nonstrict.* Traub showed that nonstrict programs can be quite difficult to compile [Tra88]. Blumofe and Leiserson [BL93] showed that some nonstrict programs have no schedule with good time and space bounds. Other work in the chess literature has focused mostly on the work-efficiency, rather than the critical path of the search. One exception is Fishburn [Fis84] who analyzes the critical path for MWF by writing a recurrence relation that expresses the critical path length, starting at end of the program and proceeding backwards to the beginning. Feldmann *et al.* [FMM91] take steps to reduce the critical path length, without having analyzed or measured it. It is difficult to determine from their papers whether their efforts are useful. Marsland *et al.* [MOS86] use overhead measurement to characterize the performance of their parallel programs. Overhead measurement is inaccurate and does not provide the performance insight that critical path and work provide. I have systematically used critical path length to estimate the average available parallelism for a chess program.

Program StarTech measures critical path lengths as an aid to understanding the performance of the program. Instead of doing an infinite processor simulation, my approach is to compute estimated critical path lengths during an actual production run on a parallel machine. In spite of the fact that the work and critical path length is not well defined for my algorithm, I found that I could tune the performance of the program on small machines and accurately predict the performance on big machines. By combining the measurement of the critical path with the measurements on large machines, I was able to validate my performance model and do most of my code-development on more readily available small machines. While many other researchers have designed multiprocessor systems to support dynamic MIMD-style programming [DR81, Hal84, Bir89, SYH*89, Cla87, PC90, CSS*91, NPA92, Hal94], those systems provide no time or space bounds. D. Culler describes some *ad hoc* techniques for managing space bounds of dataflow programs [Cul89].

The StarTech scheduling mechanisms may be useful for supporting a wide range of dynamic MIMD-style computations, including backtrack search and programs written in multithreaded programming languages.

Data Network Performance Mechanisms

With naive message-passing software and processor-network interfaces, a parallel computer may achieve only a fraction of the communications performance of the underlying network. Empirical results indicate that there are several mechanisms and techniques that can be used at various levels in the system to ensure high performance.

We found three underlying mechanisms that can improve performance. Barriers can be used to quickly determine when all processors are finished sending or when all are finished receiving. The order in which packets are injected into the network can be managed. The rate at which packets are injected into the network by the sender can be tuned to match the rate at which the target can receive messages.

There are several reasons why these mechanisms work. They can help avoid target collisions, in which several processors send to one receiver at the same time. Over time, they can help to smooth out the bandwidth demands across various bisecting cuts of the network. The mechanisms can help the programmer ensure that the packets in the network at any given time have independent, evenly distributed, destinations. These mechanisms also provide various forms of flow control, which improves the efficiency of the network. For example, barriers act as global all-pairs flow control, guaranteeing that no processor gets too far ahead at the expense of another processor.

Our results indicate that it may be a good idea to place some of our mechanisms into the network and the processor-network interface. A parallel computer should provide a rich collection of global flow-control mechanisms. Almost any form of flow control is better than none. The best way to apply each of these flow control mechanisms is not fully understood. It may be helpful to have hardware support to determine more about dynamic network congestion, in addition to the CM-5's mechanism that indicates the presence of an arrival.

A parallel computer should support fast predictable barriers. The cost of a barrier should be competitive with the cost of sending a message. The behavior of barriers should be independent of the primary traffic injected into the data network. The CM-5 provides such barriers by using a hardware global-synchronization network. Multiple priorities or logical networks could also be used. It may be beneficial for the system to perform a periodic barrier automatically to keep processors synchronized during communications operations.

The receiver must be at least as fast at the sender. Allowing user-level access to the network interface is the most important step in this direction. Hardware support to speed up the receiving of messages even by a few cycles would help improve the programmability of the CM-5, however.

The network, the processor-network interface, and its software should provide mechanisms to manage the order in which packets are injected into the network. A direct-memory-access (DMA) engine for sending and receiving packets, such as those proposed for MIT's *T [PBG*93] and Stanford's FLASH [KOH*94] machines, can make it easier to overlap communication and computation, but our experiments indicate that such engines may require fairly sophisticated packet-ordering and flow-control policies to achieve good performance. Similarly, very large packets are probably a bad choice because they have the effect of preventing packet reordering. The *entire* system must avoid head-of-line blocking.

I believe that these results apply to a wide range of parallel computers, because the observed effects are fundamental. Bandwidth considerations, scheduling issues, flow control, and composition properties will appear in any high-performance communications network. In particular, the rate at which a receiver can remove messages from the network may be the fundamental limiting issue in any network that has sufficient bandwidth to ensure that internal congestion is not the dominant issue.

Our experiments provide empirical evidence that some of the strategies used by theorists to prove theorems also make sense in real parallel systems. Theorists have argued that slowing things down can speed things up or provide predictable behavior [GL89]. Both barriers and bandwidth matching, which at some level slow down the system, actually speed things up. The use of barriers prevents processors that have gotten a little bit ahead from widening their lead. Parallel computing is not a marathon in which the first processor that finishes wins. It is a race against the clock in which we care about the finishing time of the slowest processor. Bandwidth matching is analogous to freeway on-ramp meters, which reduce the variance of the arrival rate to keep traffic flowing smoothly. Other examples of slowing things down to keep them working well include Ethernet's adaptive backoff [MB76], and the telephone system's approach to dealing with busy calls by forcing the caller to redial rather than wait in a queue [Kle76, p.103].

Not only are there sound theoretical arguments for randomized injection order [GL89, LAB93], but there is significant practical application of reordering, even when starting with what seems like a reasonable injection order.

Theorists have argued that measuring the load across cuts of a network is a good way to model performance and to design algorithms [LM88]. We have seen a few situations where we could apply such reasoning to make common patterns such as all-pairs run faster.

The MIT Strata library already incorporates these techniques for the CM-5. In the future, this research may lead to the development of mechanisms that routinely provide predictable, high-performance communication.

Synchronized MIMD Computing

The evidence in this dissertation is that synchronized MIMD hardware is a good idea. The control network, the new feature added to MIMD hardware to get synchronized MIMD hardware, allows the programmer to take advantage of frequent barriers. The control network functionality could potentially be implemented without special purpose hardware, but one requirement is that the high performance is unaffected by other users or even by the programmer's own data messages. The processors of today and of the near-future will probably not be able to handle messages quickly enough to implement a low-latency synchronization using only data messages. On machines in which when the processor-network interface requires only one instruction to handle a message, perhaps data networks will be a viable alternative to a hardware control network. Currently, the only machines that have fast enough network interfaces to support effective global synchronization directly in the data network are dataflow machines such as Monsoon [PC90] and the EM-4 [SYH*89, SKS*92], but those dataflow machines do not offer performance that is independent of the load in the data network. One of the major contributors to the latency of the control network on the CM-5 is the fact that the processor takes several microseconds to interact with the control network. The mechanisms used to make data network interfaces go faster may also be applicable to making control network interfaces go faster. It is difficult to predict what the implementation will look like, but programmers will need access to fast global synchronization.

Barriers provide a *composition* property. If you understand the performance and meaning of A and of B, then you can easily understand the performance and meaning of

$$A; \text{barrier}; B.$$

We saw that when programming with the data parallel model, some barriers that are not needed for *correctness* are still desirable for *performance.* Barriers are also useful for implementing schedulers that have a global-correctness property, such as not running out of memory, or not swamping the data network with requests for work.

The synchronized MIMD architecture effectively supports the data parallel programming model, and it can also be used to support a more general dynamic MIMD style of programming. In order to make it easier for programmers to use a dynamic MIMD style of programming, linguistic support is needed. Dataflow languages such as ID [Nik87] make it easy to write a dataflow program, but do not provide any time or space bounds for the program. In fact, Blumofe and Leiserson [BL93] showed that there are ID programs that cannot be scheduled to run quickly without using an unreasonable amount of memory. To solve these problems calls for the development of an *algorithmic* dataflow language that can guarantee time and space bounds.

Synchronized MIMD computers have the potential to relegate workstations to the status of mere terminals. Fundamentally, there is no reason to put a computer on every desk, if the user could get a fair-share of a much larger computing power. Eventually, the additional cost of having a parallel computer instead of a collection of workstations will become negligible. Users will want operating systems that guarantee that they can always access their fair share of computing power. Fair scheduling is not a major goal of today's operating system development. Perhaps some of the techniques described here will be applicable to the design of fair operating systems.

My scheduling mechanisms work well for medium-to-course grained dynamic MIMD programs. To obtain good performance from fine-grained dynamic MIMD programs may require additional work. It is not clear just how much fine-grained parallelism is going to be able to buy us, but M. Lam and R. Wilson provide one hint in their work that indicates that there is potentially more than 50-fold parallelism in ordinary C programs [LW92]. One direction to pursue this research is to try to apply my scheduling strategies to standard C programs. It has been widely argued that one cannot afford to put any new 'academic' mechanisms into state-of-the-art RISC microprocessors because of the billion-dollar investment that is put into such microprocessors. That billion-dollar investment, however, is indicative of the fact that the 'Reduced Instruction Set Complexity' designs have become very complex indeed. Any approach that simplifies the design of a microprocessor may have an overall advantage. Perhaps there is, once again, more opportunity to make gains based on architectural improvements rather than from shaving of nanoseconds.

In the future, not only parallel programs, but ordinary programs such as editors and compilers, may be able to exploit the power of large shared parallel computers. Such a goal is far from being realized, but the mechanisms described in this work may help pave the way towards that kind of truly practical parallel computing.

Acknowledgments

I would like to thank Dana Sue Henry who, as a graduate student, provided clear thinking and criticism of my work; and, as my wife, provided support, and encouragement, and took on most of the work of caring for our daughter during the first few months of her life and the last few months of my thesis work.

Anne Henry Kuszmaul, our daughter, gave that last bit of incentive to finish my thesis and get a job.

My thesis advisor and good friend, Professor Charles E. Leiserson, always challenged me to excel and to play the game at the next level of sophistication. Charles always has good ideas for improving my work and its presentation.

Professor Arvind directed my ID compiler research (which did not turn out to be part of this thesis), and patiently encouraged me to do the research that I really wanted to do.

Professor Frans Kaashoek and Professor Gregory Papadapoulos served as readers on my thesis committee.

Chapter 2 of this dissertation owes much to the coauthors of "The Network Architecture of the Connection Machine CM-5" [LAD*92]: Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak.

On behalf of my coauthors, I would like to thank the many people at Thinking Machines who contributed to the success of the CM-5 project. Space does not permit me to thank them all individually. Of special mention, however, are Pam Chulada-Smith, for keeping our systems running; Ken Crouch, for helping to design and test the data network protocols; Rick Epstein, who managed the hardware design of boards; Rolf-Dieter Fiebrich, who managed the CM-5 project; Don Moodie, who engineered the mechanical design; Sami Nuwayser, for his system administration and chip-design talents; Dave Patterson, for encouraging us to resurrect our MIMD-plus-control-network design; David Potter, who led the electrical and packaging effort; Linda Simko, whose administrative talents saved us from needing to have any; Cindy Spiller, for writing the initial system code and helping to verify the system; Guy Steele, for insights into the linguistic implications of the network architecture, and Jon Wade, who contributed to the design of the differential pads and system clocking.

In addition, I would like to thank Keira Bromberg, Dan Cassiday, Chung Der, John Devine, John Earls, Jamie Frankel, Steve Goldhaber, Harold Hubschman, Jim Lalone, Don Newman, Ralph Palmer, Gary Sabot, Soroush Shakibnia, Craig Stanfill, John Sullivan, Lew Tucker, Bob Vecchioni, Charlie Vogt and Bruce Walker for their contributions to the CM-5. We also thank the representatives from the Thinking Machines' chip vendors—Mark O'Brien, Curt Dicke, Laura Le Blanc, Jerome Li, and Shabnam Zarrinkhameh of LSI Logic; and Jerry Frenkil, Frank Kelly, Marc Corbacho, Paul Pinelle, and George Wall, of VLSI Technology, Inc.—for their help in producing the network chips.

Professor Steve Ward supported my early research on the Synchronized MIMD architecture, which was presented at my oral qualifying examination.

Dick Clayton at Thinking Machines made the decision to give me a chance to be a computer architect.

The massively parallel chess work reported in this dissertation spans three universities and one corporation, with contributions from people at Carnegie-Mellon University, the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champagne (NCSA), MIT, and Thinking Machines Corporation.

My research into massively parallel chess would not have been possible without the help provided by Hans Berliner and Chris McConnell of Carnegie-Mellon University. Hans and Chris provided me with a C-language version of Hitech, and provided much helpful advice over a long period of time. I depended heavily on their judgment and chess knowledge.

At the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champagne (NCSA): Larry Smarr and Michael Welge decided with only a few hours notice to dedicate the entire 512 node CM-5 for almost a week to running chess at the 1993 ACM International Computer Chess Championship. The CM-5 support staff at NCSA was extremely helpful. Curt Canada worked heroically during the three days before the tournament to install a new version of the operating system, get everything working, and fix an intermittent hardware problem. One of the operators at NCSA spent over an hour working with me to resolve a modem problem. The NCSA runs a first-rate facility.

At MIT: Prof. Charles E. Leiserson continually thought of ways to make the chess program better. Ryan Rifkin optimized many parts of StarTech and cleaned up the the parallel work-stealing code. James Schuyler operated StarTech at the 1993 ACM Computer Chess Championship in Indianapolis. James also performed extensive testing of StarTech during the last two weeks prior to the 1993 ACM tournament. It was a real pleasure for me not to be responsible for moving the pieces and hitting the clock. James's chess skill came especially helpful during the tournament. Hans had decided that the StarTech team should not consult with him during the tournament because it would be unfair to the other participants for him to have two horses in the race. Consequently, when StarTech got into trouble with its opening book, which had been partially garbled in transit from CMU to MIT, James ungarbled the opening. Robert D. Blumofe kept coming up with new scheduling strategies to solve the problem of obtaining linear speedup without running out of memory; Daniel Nussbaum explained to me how Alewife could handle the swamping problem. Al Vezza knew who to call at NCSA when I desperately needed machine time for the 1993 chess championship. Eric A. Brewer implemented the weighted linear regression curve-fitting program that I used.

At Thinking Machines Corporation: Mark Bromley, Roger Frye, and Kurt Thearling provided important design and programming help in getting StarTech running on the CM-5. Roger almost singlehandedly built the interface between the Hitech code and the parallel computer, while Kurt wrote the first message passing software (back in the days before active messages had been invented by Thorsten von Eicken), and Mark continually amazes me with tricks to get the last bit of performance out of the CM-5 processing node. John Mucci and David Waltz provided the managerial backing to allow Mark, Roger, and Kurt to spend some time on the project. Richard Title lent me his chess equipment (some of which I broke!) and helped to test StarTech. The staff at Thinking Machines kept the machines up and 'found' for me much machine time on the factory floor that I could use to test StarTech. The support staff was also good-natured whenever I overstressed the file servers, the CM-5 operating system, or the inhouse local area network. They never deleted my files.

Hans Berliner, Richard Karp, David Slate, and Lewis Stiller all contributed to a mini-seminar on chess held at Thinking Machines Corporation on August 12, 1991. In particular, Richard Karp suggested that I base my program on Hans Berliner's Hitech rather than GNU Chess [Cra90].

Chapter 3 of this dissertation owes much to Eric A. Brewer, who coauthored with me "How to Get Good Performance from the CM-5 Data Network" [BK94]. Eric was supported by the National Science Foundation, Grant CCR-8716884; by ARPA, Contract N00014-91-J-1698; by an equipment grant from Digital Equipment Corporation; and by grants from AT&T and IBM.

Robert Blumofe implemented most of Strata's active-message layer and block-transfer engine, and is involved in the ongoing work on bandwidth matching. Charles Leiserson and Bill Weihl suggested directions to pursue, and Charles provided the marathon analogy.

Harry F. Jordan and Burton Smith helped me find references to *The Force.*

Maria Sensale, the LCS/AI reading-room librarian, more than once obtained a copy of a paper originally published a long time ago in a far away place. Paula Mickevich and Newton Loui cheerfully, by email, renewed (again and again and again) the due date of the dozens of books and technical reports that I borrowed from the reading room.

This research was sponsored by the Advanced Research Projects Agency (DoD) through the Office of Naval Research under Grant Numbers N00014-83-K-0125, N00014-84-K-0099, N00014-91-J-1698, and N00014-92-J-1310.

This work was partially supported through the use of the Connection Machine CM-5 supercomputer at the National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign under NCSA grant number TRA930289N; and through the use of the CM-5 at the MIT Laboratory for Computer Science through project SCOUT (ARPA contract MDA972-92-J-1032.)

Bibliography

[ACD*91] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. *The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor.* Technical Report MIT/LCS/TM-454.b, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1991.

[Ahr94] Gunter Ahrendt. List of the world's most powerful computing sites. May 8, 1994. Weekly posting on newsgroup `comp.sys.super`.

[ABD82] Selim G. Akl, David T. Barnard, and Ralph J. Doran. Design and implementation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **PAMI-4** (2), pages 192–203, March 1982.

[AJ94] G. Alaghband and H. F. Jordan. Overview of the Force scientific parallel language. *Scientifc Programming,* **3** (1), pages 33–47, Spring 1994.

[AKL*88] Eugene Albert, Kathleen Knobe, John D. Lukas, and Guy L. Steele Jr. Compiling fortran 8X array features for the Connection Machine computer system. In *Parallel Programming: Experience with Applications, Languages and Systems (ACM/SIGPLAN PPEALS),* pages 42–56, ACM, New Haven, CT, July 1988. (Published as SIGPLAN Notices 29(9), September, 1988.)

[AI87] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295,* June 25-29 1987.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems,* **11** (4), pages 598–632, October 1989.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering,* **18** (3), 1992.

[BBK*68] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC-IV computer. *IEEE Transactions on Computers,* **C-17**, pages 746–757, August 1968.

[BNA91] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: extending a parallel, non-strict functional language with state. In *Functional Programming Languages and Computer Architecture (FPCA '91),* pages 538–568, Springer-Verlag, August 1991.

[BGR86] Paul D. Bassett, Lance A. Glasser, and Randy D. Rettberg. Dynamic delay adjustment: a technique for high speed asynchronous communication. In Charles E. Leiserson, ed., *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI,* pages 219–232, April 1986.

[Bat80] Kenneth E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers,* **C-29** (9), pages 836–840, September 1980.

[Bau78] G. M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* Technical Report CMU-CS-78-116, CMUCS, April 1978, 182 pp. (Ph.D. thesis.)

[BDW85] John Beetem, Monty Denneau, and Don Weingarten. The GF11 supercomputer. In *Proceedings of the 12th Annual Symposium on Computer Architecture,* pages 108–115, IEEE Computer Society Press, Boston, Massachusetts, June 1985.

[Ber79] Hans Berliner. The B* tree search algorithm: a best-first proof procedure. *Artificial Intelligence,* **12**, pages 23–40, 1979.

[BE89] Hans Berliner and Carl Ebeling. Pattern knowledge and search: the SUPREM architecture. *Artificial Intelligence,* **38** (2), pages 161–198, March 1989.

[Ber91] Hans Berliner. Personal communication. 1991.

[Ber93] Hans Berliner. Personal communication. October 1993.

[Bir80] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys,* **12** (4), pages 403–417, December 1980.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems,* **2** (1), pages 39–59, February 1984.

[Bir89] Andrew D. Birrell. *An Introduction to Progamming with Threads.* Research Report 35, DEC Systems Research Center, January 1989, 35 pp.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[BCH*93] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zhaga. *CVL: A C Vector Library.* Technical Report CMU-CS-93-114, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, February 1993, 15 pp.

[BL93] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing,* pages 362–371, San Diego, California, May 1993.

[Blu94] Robert D. Blumofe. Personal communication. 1994.

[BB94a] Robert D. Blumofe and Eric A. Brewer. Personal communication. May 16, 1994.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM,* **21** (2), pages 201–206, April 1974.

[BB94b] Eric A. Brewer and Robert D. Blumofe. *Strata: A Multi-Layer Communications Library.* Technical Report, MIT Laboratory for Computer Science, January 1994. (To appear. Available from `ftp.lcs.mit.edu` via anonymous ftp, in directory `/pub/supertech/strata`.)

[BK94] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the cm-5 data network. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS '94),* pages 858–867, April 1994.

[CK88] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing,* **2**, pages 151–169, 1988.

[Chi86] Andrew A. Chien. *Hot Spots in Routing Networks — A Collection of Studies.* Computation Structures Group Memo 267, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1986.

[Chr83] David Park Christman. *Programming the Connection Machine.* Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer

Science, January 1983. (Also available as Xerox PARC Technical Report ISL-84-3, April, 1984.)

[Cla87] T. J. W. Clarke. The D-RISC — an architecture for use in multiprocessors. In Giles Kahn, ed., *Functional Programming Languages and Computer Architectures,* pages 16–23, Springer-Verlag, Portland, Oregon, September 1987.

[CT82] J. H. Condon and K. Thompson. Belle chess hardware. In M. R. B. Clarke, ed., *Advances in Computer Chess 3,* pages 45–54, Pergamon Press, 1982. (Meeting held at the Imperial College of Science and Technology, University of London, April, 1981.)

[Cra90] Stuart Cracraft. Gnu chess. 1990. Source code available from the Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139.

[CSS*91] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV) ,* pages 164–175, Santa Clara, California, April 1991. (SIGARCH Computer Architecture News, Volume 19, Number 2, April 1991. SIGOPS Operating Systems Review, Volume 25, Special Issue, April 1991. SIGPLAN Sigplan Notices, Volume 26, Number 4, April 1991.)

[Cul89] David E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs.* Ph.D. thesis, Massachusetts Institute of Technology, 1989.

[CKP*93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Eric Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP),* pages 1–12, San Diego, California, May 1993. (ACM SIGPLAN NOTICES Volume 28, Number 7, July 1993.)

[DS87] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocesor interconnection networks. *IEEE Transactions on Computers,* **C-36** (5), pages 547–553, May 1987.

[Dal87] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, ed., *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI,* pages 391–415, The MIT Press, Cambridge, MA, Palo Alto, CA, 1987.

[DGN*86] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN.* Research Report RC 11552, Computer Sciences Department, IBM T. J. Watson Research Center, November 1986.

[DR81] J. Darlington and M. Reeve. ALICE — a multi-processor reduction machine for the parallel evaluation of applicative languages. In *Functional Programming Languages and Computer Architectures,* pages 65–76, 1981.

[DMS93] Jack J. Dongarra, Hans W. Meuer, and Erich Strohmaier. *TOP 500 Supercomputers.* Technical Report RUM 33/93, Computing Center, University of Mannheim, July 1993, i+31 pp. (Email contact: `top500@rz.uni-mannheim.de`.)

[DT90] Michel Dubois and Skreekant Thakkar. *Cache Architectures in Tightly Coupled Multiprocessors.* IEEE Computer Society, 1990. (Special Issue of *Computer*, Volume 23, Number 6.)

[Ede91] Alan Edelman. Optimal matrix transposition and bit reversal on hypercubes: all-to-all personalized communication. *J. Parallel Dist. Comp.,* **11**, pages 328–331, 1991.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985. (ACM Doctorial Dissertation Award 1985.)

[Elo78] Arpad E. Elo. *The Rating of Chessplayers — Past and Present.* Arco Publishers, New York, 1978.

[FMM90] R. Feldmann, B. Monien, and P. Mysliwietz. Distributed game tree search. In *Parallel Algorithms for Machine Intelligence and Vision,* pages 66–101, Springer-Verlag, New York, 1990.

[FMM91] R. Feldmann, P. Mysliwietz, and B. Monien. A fully distributed chess program. In D. F. Beal, ed., *Advances in Computer Chess 6,* pages 1–27, Ellis Horwood, Chichester, West Sussex, England, London, 1991. (Conference held in August 1990.)

[FMM92] R. Feldmann, P. Mysliwietz, and B. Monien. Experiments with a fully-distributed chess program. In *Heuristic Programming in Artificial Intelligence 3, The Third Computer Olympiad,* pages 72–87, Ellis Horwood, New York, 1992.

[FMM93] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. In *Advances in Computer Chess 7,* 1993. (The conference was held in June 1993, but the proceedings have not published as of August 1993.)

[FM93] Rainer Feldmann and Peter Mysliwietz. Personal communication. September 22, 1993.

[RU92] S. Felperin P. Raghavan and E. Upfal. A theory of wormhole routing in parallel computers. In *Proceedings of the 33rd Symposium on the Foundations of Computer Science,* pages 563–572, October 1992.

[FM87] Raphael Finkel and Udi Manber. DIB — a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems,* **9** (2), pages 235–256, April 1987.

[FF82] Raphael A. Finkel and John P. Fishburn. Parallelism in alpha-beta search. *Artificial Intellgence,* **19** (1), pages 89–106, September 1982.

[Fis83] John P. Fishburn. Another optimization of alpha-beta search. *SIGART Newsletter,* Number 84, pages 37–38, April 1983.

[Fis84] J. P. Fishburn. *Analysis of Speedup in Distributed Algorithms.* UMI Research Press, Ann Arbor, MI, 1984.

[Fly66] M. J. Flynn. Very high speed computing systems. *Proceedings of the IEEE,* **54** (12), pages 1901–1909, December 1966.

[Fox89] G. Fox. *Programming Concurrent Processors.* Addison Wesley, 1989.

[Gil78] James J. Gillogly. *Performance Analysis of the Technology Chess Program.* Technical Report CMU-CS-78-109, CMUCS, March 1978.

[GGK*83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU ultracomputer — designing a MIMD, shared-memory parallel machine. *IEEE Transactions on Computers,* **C-32** (2), pages 175–159, February 1983.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal,* **45**, pages 1563–1581, November 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics,* **17** (2), pages 416–429, March 1969.

[GL89] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. *Advances in Computing Research,* **5**, pages 345–374, 1989.

[GEC67] Richard D. Greenblatt, Donald E. Eastlake, III, and Stephen D. Crocker. The Greenblatt chess pgoram. In *Fall Joint Computer Conference,* pages 801–810, 1967. (reprinted as [Lev88, 56–66].)

[Gup89] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III),* pages 54–63, Boston, Massachusetts, APRIL 1989. (SIGARCH Computer Architecture News, Volume 17, Number 2, April 1989. SIGOPS Operating Systems Review, Volume 23, Special Issue, April 1989. SIGPLAN Sigplan Notices, Volume 24, Special Issue, May 1989.)

[HZJ94] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. *MIMD-Style Parallel Programming Based on Continuation-Passing Threads.* Computation Structures Group Memo 355, Massachusetts Institute of Technology, Laboratory for Computer Science, March 7, 1994, 22 pp.

[Hal94] Michael Roland S. Halbherr. *MIMD-Style Parallel Programming Based on Continuation Passing Threads.* Ph.D. thesis, Swiss Federal Institute of Technology (ETH), Department of Electrical Engineering, Zurich, May 1994. (ETH thesis Number 10699. To appear as a MIT LCS Technical Report.)

[Hal84] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming,* pages 9–17, Austin, Texas, August 1984.

[HQL*91] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems,* **3** (2), pages 377–383, July 1991.

[Hel92] Steven K. Heller. Personal communication. 1992.

[HPF93] *High Performance Fortran Language Specification Version 1.0.* High Performance Fortran Forum, May 1993.

[Hil85] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[HS86] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM,* **29** (12), pages 1170–1183, December 1986.

[HAK*] W. Daniel Hillis, Zahi S. Abuhamdeh, Bradley C. Kuszmaul, Shaw-Wen Yang, and Jon P. Wade. Digital clock buffer circuit providing controllable delay. U. S. Patent 5,118,975, issued June 2, 1992.

[HL93] Robert V. Hogg and Johanenes Ledolter. *Applied Statistics for Engineers and Physical Scientists.* Macmillan Publishing Company, New York, 1993.

[HJK*94] Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, Deborah A. Wallach, and William E. Weihl. Personal communication. May 16, 1994. .

[Hsu90] Feng-hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess.* Technical report CMU-CS-90-108, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, February 1990.

[HSN89] Robert M. Hyatt, Bruce W. Suter, and Harry L. Nelson. A parallel alpha/beta tree searching algorithm. *Parallel Computing,* **10** (3), pages 299–308, May 1989.

[Int88] *A Technical Summary of the iPSC/2 Concurrent Supercomputer.* Intel Scientific Computers, 1988.

[IEEE90] IEEE standard test access port and boundary-scan architecture. 1990. (IEEE Std 1149.1-1990.)

[Jor85] Harry F. Jordan. HEP architecture, programming and performance. In *Parallel MIMD Computation: HEP Supercomputer and Its Applications,* pages 11–40, The MIT Press, Cambridge, MA, 1985.

[Jor87] H. F. Jordan. The force. In *The Characteristics of Parallel Algorithms,* pages 395–436, The MIT Press, 1987.

[JS94] Harry F. Jordan and Burton Smith. References on the Force. June 1994. Personal Communication.

[Jor78] Harry F. Jordan. A multi-microprocessor system for finite element structural analysis. In A. K. Noor and H. G. McComb, Jr., eds., *Trends in Computerized Structural Analysis and Synthesis,* pages 21–29, Pergamon Press Ltd, 1978. (Published as a special issue of *Computers & Structures*, Volume 10, Numbers 1–2.)

[KHM87] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communication,* **35** (12), December 1987.

[KZ89] Richard M. Karp and Yanjun Zhang. On parallel evaluation of game trees. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures,* pages 409–420, Santa Fe, New Mexico, June 1989.

[Kau92] Larry Kaufman. Rate your own computer. *Computer Chess Reports,* **3** (1), pages 17–19, 1992. (Published by ICD, 21 Walt Whitman Rd., Huntington Station, NY 11746, 1-800-645-4710.)

[Kau93] Larry Kaufman. Rate your own computer — part II. *Computer Chess Reports,* **3** (2), pages 13–15, 1992-93.

[KK79] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks,* **3**, pages 267–286, 1979.

[Kle76] Leonard Kleinrock. *Queueing Systems — Volume I: Theory.* John Wiley & Sons, New York, 1976.

[Kle78] Leonard Kleinrock. Principles and lessons in packet communications. *Proceedings of the IEEE,* **66** (11), pages 1320–1329, November 1978.

[KK88] Thomas F. Knight, Jr. and Alexander Krymm. A self-terminating low-voltage swing CMOS output driver. *IEEE Journal of Solid-State Circuits,* **23** (2), pages 457–464, April 1988.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence,* **6** (4), pages 293–326, Winter 1975.

[KB82] D. Kopec and I. Bratko. The Bratko-Kopec experiment: a comparison of human and computer performance in chess. In M. R. B. Clarke, ed., *Advances in Computer Chess 3,* pages 57–72, Pergamon Press, 1982. (Meeting held at the Imperial College of Science and Technology, University of London, April, 1981.)

[KOH*94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Koroush Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford FLASH multiprocessor. In *The 21st Annual International Symposium on Computer Architecture,* pages 302–313, Chicago, Illinois, April 18–21, 1994.

[KTR93] T. T. Kwan, B. K. Totty, and D. A. Reed. Communication and computation performance of the CM-5. In *Proceedings of Supercomputing '93,* pages 192–201, November 1993.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The 19th Annual International Symposium on Computer Architecture,* pages 46–57, Gold Coast, Australia, May 1992. (ACM SIGARCH Computer Architecture News, Volume 20, Number 2.)

[Las85] Clifford Adam Lasser. *SIMPL, A Programming Language for the Connection Machine.* Bachelor's thesis, MIT Department of Electrical Engineering and Computer Science, 1985.

[LMR88] F. T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *29th Annual IEEE Symposium on Foundations of Computer Science,* pages 256–271, 1988.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers,* **C-34** (10), pages 892–901, October 1985.

[LM88] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica,* **3**, pages 53–77, 1988.

[LAD*92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms '92,* pages 272–285, San Diego, California, June 1992.

[Lev88] David Levy. *Computer Chess Compendium.* W. Turner & Son, Ltd (Distributed in the USA through Springer-Verlag, New York), 1988.

[Lim91] Beng-Hong Lim. *Waiting Algorithms for Synchronization in Large-Scale Multiprocessors.* Technical Report MIT/LCS/TR-498, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1991, 103 pp. (Master's thesis.)

[LT77] G. J. Lipovski and A. Tripathi. A reconfigurable varistructure array processor. In Jean-Loup Baer, ed., *Proceedings of the 1977 International Conference on Parallel Processing,* pages 165–174, IEEE Computer Society, August 23–26, 1977.

[Liu68] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill Book Company, New York, 1968.

[LAB93] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An atomic model for message-passing. December 1993. given to cel by the authors.

[LB80] Stephen F. Lundstrom and George H. Barnes. A controllable MIMD architecture. In *Proceedings of the 1980 International Conference on Parallel Processing,* pages 19–27, August 1980.

[MR74] T. A. Marsland and P. G. Rushton. A study of techniques for game-playing programs. In *Advances in Cybernetics and Systems,* pages 363–371, Gordon and Breach, London, 1974.

[MC82] T. A. Marsland and M. S. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys,* **14** (4), pages 533–552, December 1982.

[MOS86] T. A. Marsland, M. Olafsson, and J. Schaeffer. Multiprocessor tree-search experiments. In D. F. Beal, ed., *Advances in Computer Chess 4,* pages 37–51, Pergamon Press, Meeting held at Brunel University, London, April, 1984, 1986.

[McA88] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence,* **35**, pages 287–310, 1988.

[MR90] Michael Metcalf and John Ker Reid. *FORTRAN 90 Explained.* Oxford University Press, New York, 1990.

[MB76] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *IEEE Transactions on Computers,* **19** (7), pages 395–404, July 1976.

[MKH91] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Trans. on Parallel and Distributed*

Systems, **2** (3), pages 264–280, July 1991. (A slightly more complete version appears as technical report MIT/LCS/TM-449, June 1991.)

[MS91] Jacek Myczkowski and Guy Steele. Seismic modeling at 14 gigaflops on the connection machine. In *Supercomputing '91,* pages 316–326, Albuquerque, NM, November 1991.

[New75] Monroe Newborn. *Computer Chess.* Academic Press, New York, 1975.

[Nik87] *Id Nouveau Reference Manual.* Massachusetts Institute of Technology, Laboratory for Computer Science, Computations Structures Group, April 24, 1987.

[NFH88] Rishiyur S. Nikhil, P. R. Fenstermacher, and J. E. Hicks. *Id World Reference Manual (for LISP Machines).* Unnumbered Technical Report, Massachusetts Institute of Technology, Laboratory for Computer Science, Computations Structures Group, August 23, 1988. (Supersedes Dinarte R. Morais, Id World: User's Manual, Computations Structures Group Memo 266, June 1986.)

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *The 16th Annual International Symposium on Computer Architecture,* pages 262–272, Jerusalem, Israel, May 28–June 1, 1989. (ACM SIGARCH Computer Architecture News, Volume 17, Number 3, June 1989.)

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *t: a multithreaded massively parallel architecture. In *The 19th Annual International Symposium on Computer Architecture,* pages 156–167, Gold Coast, Australia, May 1992. (ACM SIGARCH Computer Architecture News, Volume 20, Number 2.)

[Nus93] Daniel Nussbaum. *Run-Time Thread Management for Large-Scale Distributed-Memory Multiprocessors.* Ph.D. thesis, Massachusetts Institute of Technology, September 1993.

[OD90] Mathew T. O'Keefe and Henry G. Dietz. *Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM).* Technical Report TR-EE 90-9, School of Electrical Engineering, Purdue University, January 1990.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token store architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA,* May 1990.

[PBG*93] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. Integrated building blocks for parallel computers. In *Proceedings of Supercomputing '93,* pages 624–635, November 1993.

[PS91] W. Paul and D. Scheerer. The DATIS-P fault tolerant machine. In *Proc. 24th HICSS (Hawaii Int. Conf. on System Sciences),* pages 560–571, 1991.

[Pea80] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence,* **14** (2), pages 113–138, September 1980.

[Per87] R.H. Perrott. *Parallel Programming.* Addison Wesley, Reading, MA, USA, 1987.

[PN85] G. F. Pfister and V. A. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers,* **C-34**, pages 943–948, October 1985.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science,* **5**, pages 223–255, 1977.

[Pol88] Constantine D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers,* **37** (8), pages 991–1004, August 1988.

[PM83] F. Popowich and T. A. Marsland. *Parabelle: Experience with a Parallel Chess Program.* Technical Report 83-7, Computing Science Department, University of Alberta, Edmonton, Canada, 1983.

[PTV*92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge Univeristy Press, New York, 1992.

[RBJ86] Abihram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The fluent abstract machine. In Charles E. Leiserson, ed., *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI,* pages 71–93, April 1986. (Also published as Yale University Tech. Report YALEU/DCS/TR-573, January 1988.)

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* pages 69–80, Portland, Oregon, June 21–23, 1989.

[RS87] John R. Rose and Guy L. Steele Jr. *C*: An Extended C Language for Data Parallel Programming.* Technical Report PL87-5, Thinking Machines, Inc., April 1987.

[Sab88] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, Cambridge, MA, 1988.

[SYH*89] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel,* pages 46–53, May 28-June 1, 1989.

[Sch93] Robert Schumacher. *One Year KSR1 at the University of Mannheim Results and Experiences.* Technical Report RUM 35/93, University of Mannheim, December 1993, 67 pp. (Available from `ftp.uni-mannheim.de` via anonymous ftp, in directory `info/rumdoc`. The files `rum3593.ps` and `rum3593s.ps`, are 300dpi and 400dpi respectively.)

[Sei85] C. L. Seitz. The cosmic cube. *Communications of the ACM,* **28** (1), pages 22–23, January 1985.

[SAD*86] Charles L. Seitz, William C. Athas, William J. Dally, Reese Faucette, Alain J. Martin, Sven Mattisson, Craig S. Steele, and Wen-King Su. *Message-Passing Concurrent Computers: Their Architecture and Programming.* Addison-Wesley, Reading, MA, 1986.

[SW75] Stephen D. Senturia and Bruce D. Wedlock. *Electronic Circuits and Applications.* John Wiley & Sons, New York, 1975.

[Sha50] Claude Shannon. Programming a digital computer to play chess. *The Philosphical Magazine,* **41 - 7th Series** (314), pages 256–275, March 1950. (First presented at the National IRE Convention, March 9, 1949, New York, U.S.A. Reprinted in *The World of Mathematics*, Newman (ed.), Volume 4, Simon and Schuster, 1954, New York, and reprinted in [Lev88, pp. 2–13].)

[SKS*92] A. Shaw, Y. Kodama, M. Sato, S. Sakai, and Y. Yamaguchi. Performance of data-parallel primitives on the EM-4 dataflow parallel supercomputer. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation,* pages 302–309, October 1992.

[SKS81] Howard Jay Siegel, Frederick C. Kemmerer, and Harold E. Smalley, Jr. PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers,* **C-30** (12), pages 934–947, December 1981.

[SA77] David J. Slate and Lawrence R. Atkin. CHESS 4.5 — the Northwester University chess program. In *Chess Skill in Man and Machine,* pages 82–118, Springer-Verlag, New York, 1977. (Reprinted as [SA83] and in [Lev88, pp. 80–103].)

[SA83] David J. Slate and Lawrence R. Atkin. CHESS 4.5 — the Northwestern University chess program. In *Chess Skill in Man and Machine,* pages 82–118, Springer-Verlag, 1983.

[Smi94] Burton Smith. Personal communication. May 15, 1994.

[SYC92] M. St. Pierre, S.-W. Yang, and D. Cassiday. Functional VLSI design verification methodology for the CM-5 massively parallel supercomputer. In *International Conference on Computer Design,* October 1992.

[Ste90] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages,* pages 218–231, Association for Computing Machinery, San Francisco, CA, January 1990.

[Ste92] Clifford Stein. Evaluating game trees in parallel. In Charles E. Leiserson, ed., *Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems,* pages (47-1)–(47-2), MIT Endicott House, July 1992.

[Sto79] G. C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence,* **12** (2), pages 179–196, August 1979.

[SS82] Salvatore J. Stolfo and David Elliot Shaw. DADO: a tree-structured machine architecture for production systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-82),* pages 242–245, Pittsburgh, PA, 18–20, 1982. (.)

[Tv85] Andrew S. Tanenbaum and Robbert van Rennesse. Distributed operating systems. *ACM Computing Surveys,* **17** (4), pages 419–470, December 1985.

[TY86] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, eds., *Proceedings of the 1986 International Conference on Parallel Processing,* pages 528–535, August 1986.

[TI90] *SN54ACT8997, SN74ACT8997 Scan Path Linker With 4-bit Identification Bus.* Texas Instruments, April 1990.

[TL93] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems,* **11** (2), pages 179–203, May 1993.

[Thi86a] *The Essential* STARLISP *Manual.* Thinking Machines Corporation, July 1986.

[Thi86b] *Connection Machine Parallel Instruction Set (PARIS), Release 2, Revision 7.* Thinking Machines Corporation, July 1986.

[TMC88] Method and apparatus for aligning the operation of a plurality of processors. Thinking Machines Corporation, applicant. W. Daniel Hillis, inventor. European Patent Application Serial Number 89 902 461.6, priority date of February 2, 1988. Also International Application Number WO 89/07299 (Published under the Patent Cooperation Treaty), Publication Date 10 August 1989.

[TMC91] *The Connection Machine CM-5 Technical Summary.* Thinking Machines Corporation, October 1991.

[Thi93] *CMMD User's Guide.* Thinking Machines Corporation, 1993.

[Tho83] Clark D. Thompson. The VLSI complexity of sorting. *IEEE Transactions on Computers,* **C-32**, pages 1171–1184, December 1983.

[Tra88] Kenneth R. Traub. *Sequential Implementation of Lenient Programming Languages.* Technical report MIT/LCS/TR-417, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1988. (Ph.D. thesis.)

[Tru81] T. R. Truscott. *Techniques Used in Minimax Game-Playing Programs.* Master's thesis, Computer Science Department, Duke University, Durham, NC, April 1981.

[Tru92] Tom Truscott. Personal communication. September 23, 1992.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software-Practice and Experience,* **9**, pages 32–49, 1979.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM,* **33** (8), pages 103–111, August 1990.

[vCG*92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture,* pages 256–266, Gold Coast, Australia, May 1992. (ACM SIGARCH Computer Architecture News, Volume 20, Number 2.)

[YTL87] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-stop addressing in large scale multiprocessors. *IEEE Transactions on Computers,* **C-36** (4), pages 388–395, April 1987.

[ZH92] Robert Zak and Jeffrey Hill. An IEEE 1149.1 compliant testability architecture with internal scan. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors,* October 1992.

[ZBG88] H. P. Zima, H.-J. Bast, and M. Gerndt. SUPERB—a tool for semiautomatic MIMD/SIMD parallelization . *Parallel Computing,* **6**, pages 1–18, 1988.

[Zob70] A. L. Zobrist. *A Hashing Method with Applications for Game Playing.* Technical Report 88, Computer Science Department, University of Wisconsin, Madison, April 1970.

Biographical Note

Bradley C. Kuszmaul was born in Iowa City, Iowa, and raised in Merriam, Kansas. Bradley is married to Dana Sue Henry, who is also a graduate student studying computer architecture at MIT. Bradley and Dana have a daughter, Anne Henry Kuszmaul.

Bradley earned two Bachelor of Science degrees, one in mathematics and one in computer science and engineering, from the Massachusetts Institute of Technology in 1984. His bachelor's thesis in computer science, *Type Checking in VIM-VAL*, was runner-up for the William A. Martin Prize for the best undergraduate thesis in computer science, May 1984. Bradley entered the doctoral program at MIT, and in 1986 obtained a Master of Science degree in electrical engineering and computer science with a master's thesis titled *Simulating Applicative Architectures on the Connection Machine*. Bradley's parallel computer chess program, StarTech, running on a 512 processor CM-5, tied for Third Prize in the 1993 ACM International Computer Chess Championship, held in Indianapolis, Indiana, February 14–17, 1993.

Bradley has been a consultant for Thinking Machines Corporation since January 1984, when he wrote the test vectors for the CM-1 chip. Bradley's experience writing diagnostics for the CM-1 and CM-2, as well as helping with the design and implementation of the first *LISP interpreter, served him well during 1987–88, when he spent 15 months working full-time at Thinking Machines Corporation as one of the principal architects of the Connection Machine CM-5 supercomputer. Before 1984 Bradley worked summers as a freelance software engineer.

Bradley has served as a teaching assistant for several classes at MIT, including 6.033, *Computer System Engineering;* 6.823, the graduate core course *Computer System Architecture;* 6.004, *Computation Structures;* 6.170, *Software Engineering;* and has tutored undergraduate mathematics and physics in the MIT Experimental Study Group.

Bradley has been an inventor or coinventor on several patents, including

David C. Douglas, W. Daniel Hillis, Bradley C. Kuszmaul, Charles E. Leiserson, Shaw-Wen Yang, and Robert C. Zak. Parallel computer system including arrangement for transferring messages from a source processor to selected ones of a plurality of destination processors and combining responses. U. S. Patent, Issued Summer 1993.

Bradley C. Kuszmaul. Method of routing a plurality of messages in a multi-node computer network. U. S. Patent 5,111,198, issued May 5, 1992.

W. Daniel Hillis, Zahi S. Abuhamdeh, Bradley C. Kuszmaul, Shaw-Wen Yang, and Jon P. Wade. Digital clock buffer circuit providing controllable delay. U. S. Patent 5,118,975, issued June 2, 1992.

David C. Douglas, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Charles E. Leiserson, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and

Robert C. Zak. Parallel Computer System. International patent application published under the patent cooperation treaty. International Publication Number WO 92/06436. International Publication Date 16 April 1992.

Publications in refereed journals include

Bradley C. Kuszmaul. Fast, deterministic routing, on hypercubes, using small buffers. *IEEE Transactions on Computing*, **39** (11), pages 1390–1393, November 1990.

Bradley C. Kuszmaul and Jeff Fried. NAP (No ALU Processor): The great communicator. *Journal of Parallel and Distributed Computing*, **8** (2), pages 169–179, February 1990. (An early version of this paper appeared as Bradley C. Kuszmaul and Jeff Fried. NAP (No ALU Processor): The great communicator. In *Frontiers of Massively Parallel Computation. Proceedings of the 2nd Symposium,* Fairfax, VA, October 10–12 1988.)

Publications (not mentioned above) in refereed conferences include

Bradley C. Kuszmaul. A glitch in the theory of delay-insensitive circuits. In *1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, The University of British Columbia, Vancouver, British Columbia, Canada, August 1990.

Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures (SPAA '92)*, pages 272–285, June 1992.

Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *International Parallel Processing Symposium (IPPS '94)*, pages 858–867, April 1994.