The StarTech Massively Parallel Chess Program

Bradley C. Kuszmaul
bradley@lcs.mit.edu
http://theory.lcs.mit.edu/~bradley

Laboratory for Computer Science
Massachusetts Institute of Technology
NE43-247, 545 Technology Square
Cambridge, MA 02139

January 11, 1995

Abstract

The StarTech massively parallel chess program, running on a 512-processor Connection Machine CM-5 supercomputer, tied for third place at the 1993 ACM International Computer Chess Championship. StarTech employs the Jamboree search algorithm, a natural extension of J. Pearl's Scout search algorithm, to find parallelism in game-tree searches. StarTech's work-stealing scheduler distributes the work specified by the search algorithm across the processors of the CM-5. StarTech uses a global transposition table shared among the processors. StarTech has an informally estimated rating of over 2400 USCF.

Two performance measures help in understanding the performance of the StarTech program: the work, W, and the critical path length, C. The Jamboree search algorithm used in StarTech seems to perform about 2 to 3 times more work than does our best serial implementation. The critical path length, under tournament conditions, is less than 0.1% of the total work, yielding an average parallelism of over 1000. The StarTech scheduler achieves actual performance of approximately $T \approx 1.02W/P + 1.5C$ on P processors. The critical path and work can be used to tune performance by allowing development of the program on a small, readily accessable, machine while predicting the performance on a big, tournament-sized, machine.

1 Introduction

Computer chess provides a good testbed for understanding dynamic multithreaded computations. The parallelism in computer chess is derived from a dynamic expansion of a highly irregular game-tree, which has historically made it difficult to implement parallel computer chess programs and other dynamic applications. To investigate how to program such applications, I engineered a parallel chess program called StarTech (pronounced "Star-Tek") on the Connection Machine CM-5. Even though my primary area of research is in parallel computing rather than computer chess, the StarTech project has produced some interesting computer chess technology. This paper explains how StarTech works and how it performs.

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 program [BE89]. Hitech 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 me with an implementation of Hitech written in C (without any search extensions except for quiescence with check extension) that runs at 2,000 to 5,000 positions per second. I built a parallel program using Berliner's serial code and reimplemented parts of the serial program to make it faster. Both the serial and parallel versions of my program are called StarTech. StarTech, unlike Hitech, does not use the null-move search, which probably costs StarTetch about a factor of two in performance. StarTech uses the same search extensions in both the serial and the parallel implementations.

I divided the programming problem into two parts: an application and a scheduler. The application can be thought of as a dynamically unfolding tree of chess positions. There are dependencies among the positions. A position may not be searched until the positions it depends on have been searched. The application specifies the shape of the tree and the dependencies between the positions. The scheduler, on the other hand, takes such an application and decides on which processor each position should be evaluated, and when the evaluation should be done. The application's job is to expose parallelism. The scheduler's job is to run the program as fast as possible, given the parallelism in the application. Thus, StarTech is conceptually divided into two parts: The parallel game tree search algorithm (the application), which specifies *what* can be done in parallel; and the scheduler, which specifies *when* and *where* the work will actually be performed.

The remainder of this paper is organized as follows. Section 2 describes how StarTech works explaining the Jamboree game-tree search algorithm and sketching the implementation of the global transposition table. A performance

This research was supported in part by the Advanced Research Projects Agency (DoD) under Grants N00014-94-1-0985, N00014-92-J-1310, and N00014-91-J-1698.

study, relating StarTech's performance to two fundamental measures of parallel performance, is presented in Section 3. Section 4 shows how I used those metrics to improve the performance of StarTech. Section 5 concludes with some remarks on the challenges of parallel computer chess.

2 How StarTech Works

Before examining the performance of StarTech in Section 3, this section explains how StarTech works. First, an overall description of how StarTech searches in parallel is presented. Then, the Jamboree algorithm is explained, starting with a review of the serial Scout search algorithm. Next, the board-representation problem, which is faced by the implementors of any parallel chess program, is illustrated by showing how the repeated-position test is implemented in StarTech. Finally the implementation of StarTech's global transposition table is explained.

StarTech's game-tree search algorithm is 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 usually succeed, because the move ordering is often best-first due to 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 [FMM93]. Still others have proposed or analyzed variations of this style of game tree search [ABD82, MC82, Fis84, Hsu90]. My Ph.D. thesis [Kus94] provides a more complete discussion of how Jamboree search is related to other search algorithms. I do not claim that Jamboree search is an entirely novel search algorithm, although some of the details of my algorithm are quite different from the details of related algorithms. Instead, I view the algorithm as a good testbed for understanding how to design scalable, predictable, multithreaded programs.

To distribute work among the CM-5 processors, StarTech uses a randomized 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 runs out of work locally, it sends a message to another processor, selected at random, and removes work from that processor's collection of posted work. The CM-5 has sufficient interprocessor communications performance that there is no appreciable advantage in trying to steal locally rather than from a random processor, and the randomized approach to scheduling is provably efficient [BL94].

Scout Search

Before delving into the details of Jamboree search, let us review the serial Scout search algorithm. For a parallel chess program, one needs an algorithm that both effectively prunes the tree and can be parallelized. I started with a variant on serial α-β search, called *Scout* search, and modified it to be a parallel algorithm.

Figure 1 shows the serial Scout search algorithm. (Many chess researchers refer to the Scout algorithm as "PV Search", but it appears that J. Pearl's "Scout" terminology takes precedence [Pea80].) Procedure `scout` is similar to the familiar α-β search algorithm which takes paramaters α and β used to prune the search [KM75]. The Scout algorithm, however, when considering any child that is not the first child, first performs a *test* of the child 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 idea is that testing a position is cheaper than determining 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 $(-\alpha)$ 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 is 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.

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 the parent might return immediately with the wrong answer when the parent performs the check of the returned

```
(S1)   Define scout$(n, \alpha, \beta)$ as
(S2)       If $n$ is a leaf then return static_eval$(n)$.
(S3)       Let  $\vec{c} \leftarrow$ the children of $n$, and
(S4)          $b \leftarrow -$scout$(c_0, -\beta, -\alpha)$.                         ;; Value
(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 $s \leftarrow -$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 $s \leftarrow -$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 1: Algorithm scout.

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] justifies its dominance as the search algorithm of choice in modern serial chess-playing programs.

Jamboree Search

The Jamboree algorithm, shown in Figure 2, is a parallelized version of 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. The Jamboree algorithm sequentializes the full-window searches for values, because, whereas 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.

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 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.

Parallel search of game-trees is difficult because the most efficient algorithms for game-tree search are inherently serial. We obtain parallelism by performing the tests in parallel, but those tests may not all be necessary in a serial execution order. In order to get any parallelism, we must take the risk of performing extra work that a good serial program would avoid. By taking our chances with the tests rather than the valuations, we minimize the risk of performing a huge amount of wasted work.

Copying Data for Parallel Search

Most serial chess programs use data structures that make them difficult to parallelize. For example, a typical serial program uses many global variables. Every time a subsearch is started the global variables are modified, and when the subsearch finishes, the modifications to the global variables are undone. This approach efficiently supports board representations that are large, as long as relatively few bytes change on any given move. It can be difficult to parallelize a program written in this style, however, since when a steal-request arrives, there is no explicit representation of the boards that need to be stolen from the middle of the search tree.

StarTech addresses this problem by copying the board

```
(J1)   Define jamboree$(n, \alpha, \beta)$ 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 $i$th iteration of the parallel loop.
(J19)           enddo
(J20)         return $b$.
```

Figure 2: Algorithm jamboree.

state when a subsearch is started. Thus, when a child completes, the parent still has its original, unmodified, copy of the board, and therefore no "unmodify" needs to be done. When the board is copied, however, every byte of the board representation must be copied, whether it is modified or not.

The board state can include some things that one might not expect. Consider the problem of detecting repeated positions. Most serial programs use a hash table that stores all the positions in the game history and the variation leading to a particular position. The hash table can be incrementally modified and unmodified. Somehow the set of previous positions must be represented in the board state.

StarTech represents the positions in the variation with an array of hash keys (one key for each position.) This array of positions can be thought of as the part of the board-state. (The entire game is broadcast to all the processors, so that the processors are able to examine the positions in the actual game without copying those positions repeatedly.) When a position is stolen, the sequence of positions between the root of the search tree and the stolen position is sent through the data network to the stealing processor.

The cost of copying the move history is not as great as one might expect. Only the hashes of the positions encountered since the last irreversible move need to be copied. In StarTech, the average length of the repeated-position history that is actually copied is less than one, partly because in quiescence search most of the moves are irreversible.

The Global Transposition Table

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.

Program StarTech uses a *global* transposition table distributed across all the nodes of the machine, as shown in Figure 3. To access the transposition table, which requires communicating from one node to another, a message-passing 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 processor needs to look up a position in the table, it sends a message to the processor named by the hash key, and that processor responds with a message containing the contents of the table entry. (A sim-

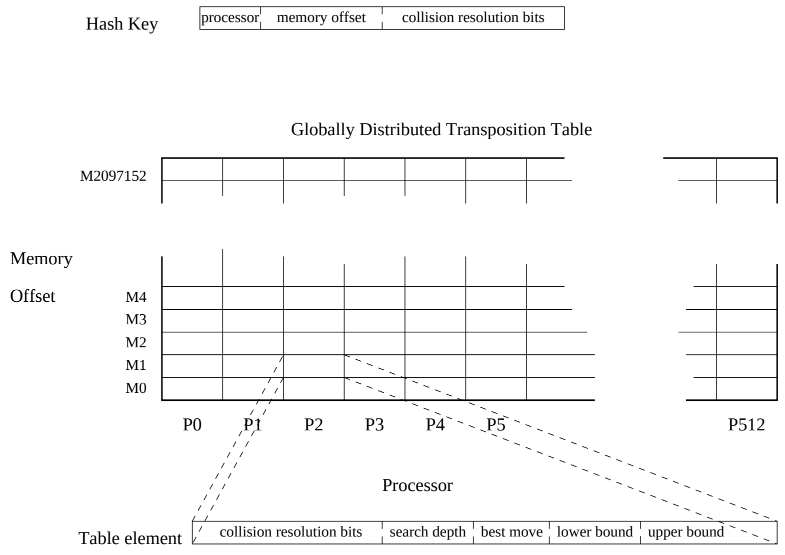

Figure 3: The Startech transposition table is globally distributed. The hash key for a position is divided into a processor number, memory offset, and collision resolution bits. The processor number and memory offset identify a unique table element in the machine. The collision resolution bits discriminate 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.

ilar transposition table scheme is used by the Zugzwang parallel chess program [FMM93].)

Historically parallel chess programs have often avoided global transposition tables. 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 (10 microsecond) messages on the CM-5. (Similarly, Cray Blitz uses a global transposition table because accessing global memory is also inexpensive on a Cray supercomputer [HSN89].)

3 The Performance of StarTech

The previous section explained how StarTech works. This section explores the performance of StarTech. We start by estimating StarTech's rating using a ratings estimation benchmark. Then, by using two fundamental parallel performance metrics, the work and the critical path length, we gain a deeper understanding of StarTech's performance. Finally, we present a few analytical results on the Jamboree search algorithm.

Estimating StarTech's Rating

The most common questions about StarTech's performance are "What is StarTech's Rating?" and "How much real performance improvement does StarTech get by using more processors?" This section attempts to answer those questions. The standard way to determine the rating of a chess player is to play lots of games. Since playing games is very time consuming, I use a set of benchmark problems designed by I.M. L. Kaufman [Kau92, Kau93] to estimate StarTech's performance using the Elo rating system [Elo78]. Kaufman cautions against misuse of his ratings estimator, for example by tuning a program to do well against only the benchmark problems. Since StarTech has not been tuned against Kaufman's benchmark, we can get some idea of StarTech's rating by using Kaufman's estimator.

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

$$\text{USCF Rating} \approx 2930 - 200 \cdot \log_{10} t_{20}, \qquad (1)$$

which means that a factor of 10 in performance is estimated to be worth 200 ratings points. In addition to looking at

Trans-position 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 4: 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 1.

Processors	Time for Top 20 (seconds)	Estimated 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 5: 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 1), and the time to solve all 25 positions.

the fast 20 positions, I also find it interesting to look at the sum of the times for all 25 positions.

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.

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. My strategy is to choose a transposition table size that is sufficiently large that increasing it further doesn't help the performance on this benchmark. Figure 4 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.

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 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. Each configuration includes a transposition table of with at least 2^{26} entries total. 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, I 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 5 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 512-nodes.

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.* [FMM93] 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 was specifically designed to estimate the rating of a program that plays master-level chess.

Even Kaufman's problem set is not difficult enough to test a program like StarTech. Partly this is because StarTech inherits HiTech's strategy of analyzing a position for a few seconds before starting the search, and partly because it takes the parallel search routine a few seconds to expose any parallelism. StarTech wants problems that will take more than just a few seconds to solve. On the 512-processor run, for many of Kaufman's positions, the program spends only a few seconds on the position. 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.) The positions in Kaufman's test that achieve the best speedup are often discarded by Kaufman's evaluation scheme because they were among the slowest five positions. In tournament play, 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. Under such conditions, StarTech seems to achieve a factor of between 50 and 100 speedup on 512 processors.

The authors of the Zugzwang chess program [FMM93] encountered similar problems, finding 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.

In summary, I found it difficult to obtain a clear estimate of the strength of StarTech from these benchmarks, but it appears safe to say that StarTech's rating would be over 2400 USCF. An effort needs to be made to find harder problems to test parallel programs.

Critical Path and Work of StarTech

Simply measuring the runtime of StarTech does not provide much insight into why the program behaves as it does. We now examine how to use two fundamental performance metrics, critical path and work, to better understand the performance of StarTech.

It was not clear to me, when I started programming StarTech, how to predict the performance of a parallel chess program. Chess programs search a dynamically generated tree, and obtain their parallelism from that tree. Different branches of the tree have vastly different amounts of total work and average parallelism. Chess programs use large global data structures and are nondeterministic. I wanted predictable performance. For example, if I develop a program on a small machine, I would like to be able to instrument the program and predict how fast it will run on a big machine. How can predictable performance be salvaged from a program with these characteristics?

I found that two numbers, the critical path length and the 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 on the runtime of the program. It turns out that C can be measured as the program runs, by a method of timestamping, without actually performing an infinite-processor simulation. The 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*. Both C and W/P are lower bounds to the runtime on P processors. (Another way to think about C and W is to consider the program to be a dataflow graph. C is the depth of the graph, and W is the size of the graph.) We can compare W to T_S, the runtime of a corresponding serial chess program, and we can compare C to W. The ratio T_S/W is the *efficiency* of the parallel program, and indicates how much overhead is inherent in the parallel algorithm. The ratio W/C is the *average parallelism* of the program, and indicates how many processors we can hope to effectively use. A good application keeps W and C small. Usually, I simply measure C and W as the program runs. The values for C and W can also be derived analytically for a few special cases.

For many non chess applications the values of W and C depend only on the application, rather than on the scheduler. In StarTech's search algorithm, however, the values of W and C are partially dependent on decisions made by the scheduler. I found that W and C seem to be, in practice, mostly independent of those decisions.

It is easy to measure the work of a program. I measured, using the microsecond-accurate timers of the CM-5, the total number of cycles spent running chess code on each processor. Figure 6 shows how the measured work varies with the machine size for each of several executions of each of the 25 chess positions. The time for solving the problem on my best serial version of StarTech is also shown. Note that the amount of work increases (that is, the efficiency drops) as the machine size grows. (It turns out that StarTech always runs faster on a big machine than on a small machine, however.) Whereas the efficiency drops as the machine gets larger, if we fix the machine size, and let problems run longer, the efficiency improves. (The longer running positions are shown first, and they are the positions on which less extra work is done by bigger machines.) Jamboree search achieves efficiencies of between 33% and 50%.

The critical path length is a little bit more difficult to measure. Using the CM-5 timers, I measured the length of the longest dependency chain of each tree, as the search runs. (The measurement is performed as follows: Each position's *critical-path depth* is defined to be the maximum of the critical path depths of the positions it depends on, plus the time it takes to do move generation and static evaluation on that position.) Figure 7 shows how the measured critical path length varies with the machine size for each of 25 different chess positions. The critical path also varies with the machine size because search algorithm interacts with the scheduler.

The critical path and work can actually be used to predict the performance of StarTech. I found that the run-time on P processors of StarTech is accurately modeled as

$$T_P \approx 1.02\frac{W}{P} + 1.5C + 4.3 \text{ seconds.} \qquad (2)$$

Except for the constant term of 4.3 seconds, this estimate is within a factor of 2.52 of the lower bounds given by C and W/P. The 4.3-second constant term comes from the time StarTech spends at the beginning of every search analyzing the board and initializing the evaluation tables.

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 steal 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.

Analysis of Jamboree Search

The Jamboree search algorithm can be analyzed for a few special cases of trees of uniform height and degree. It turns out that I have two analytical results, one for best ordered trees and one for worst ordered trees. The complete statement of the theorems and proofs can be found in my Ph.D. thesis [Kus94].

Theorem 1 states how Jamboree search behaves on best-ordered trees. A best-ordered tree is one in which it turns out that the first move considered is always the best move, and thus the tests in the jamboree search algorithm always succeed.

Theorem 1 *For uniform best ordered trees of degree d and height h, the efficiency is 1, and the average parallelism is about $(d/2)^{(h/2)}$.* ∎

Chess trees typically have degree between 30 and 40 in the middle-game, and which means that on a full-width search to depth 10, a best-ordered chess tree would have several hundred-thousand fold parallelism.

If the tree is not best-ordered, then the performance of the parallel algorithm can be much worse, however. Theorem 2 addresses worst-ordered trees. A worst-ordered tree is one in which the worst move is considered first, and the second worst move is considered second, and so on, with the best move considered last.

Theorem 2 *For uniform worst-ordered trees of degree d and height h, the efficiency is about $1/3$ and the average parallelism is about 3, and the speedup is always less than 1.* ∎

Surprisingly, for worst-ordered uniform game trees, the speedup of Jamboree search over serial α-β search turns out to be under 1. That is, Jamboree search is worse than serial α-β search, even on an "ideal" machine with no overhead for communications or scheduling. For comparison, parallelized negamax search achieves linear speedup on worst-ordered trees, and Fishburn's MWF algorithm achieves nearly linear speedup on worst-ordered trees [Fis84].

In summary, critical path and work are the important parameters for understanding the performance of StarTech. The average parallelism and efficiency of StarTech are both good enough to achieve significant speedup on chess problems, which probably allows StarTech to performs at Senior Master level.

4 Improving StarTech

We have seen how StarTech works, and some basic performance characteristics of the program. This section shows how the critical path and work can be used to improve the performance of StarTech. First we look at a traditional profile of how time is spent by StarTech, and then we review three strategies that I found can improve the performance of the program.

How Time is Spent in StarTech

Examining a timing profile can provide clues for how to improve a program. Figure 8 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 biggest chunk of time is devoted to the chess-work, which further broken down in Figure 9.

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 my 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.

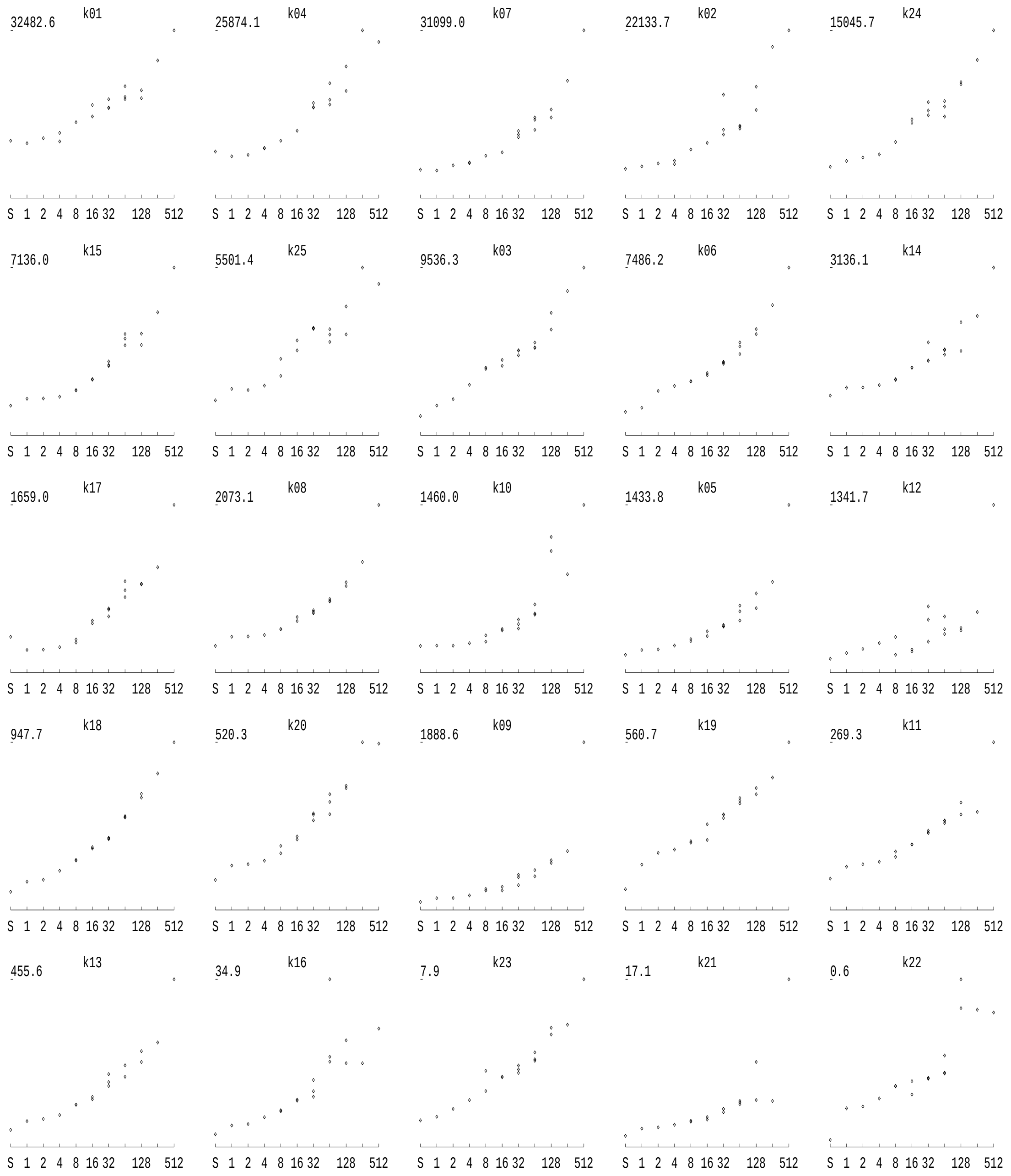

Figure 6: The total work of each of Kaufman's 25 test positions, as measured on various machine sizes. Each box represents one test position. The positions are named `k01` through `k25`. The horizontal axis on each graph is the machine size, where 'S' denotes my best serial implementation. The vertical 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.

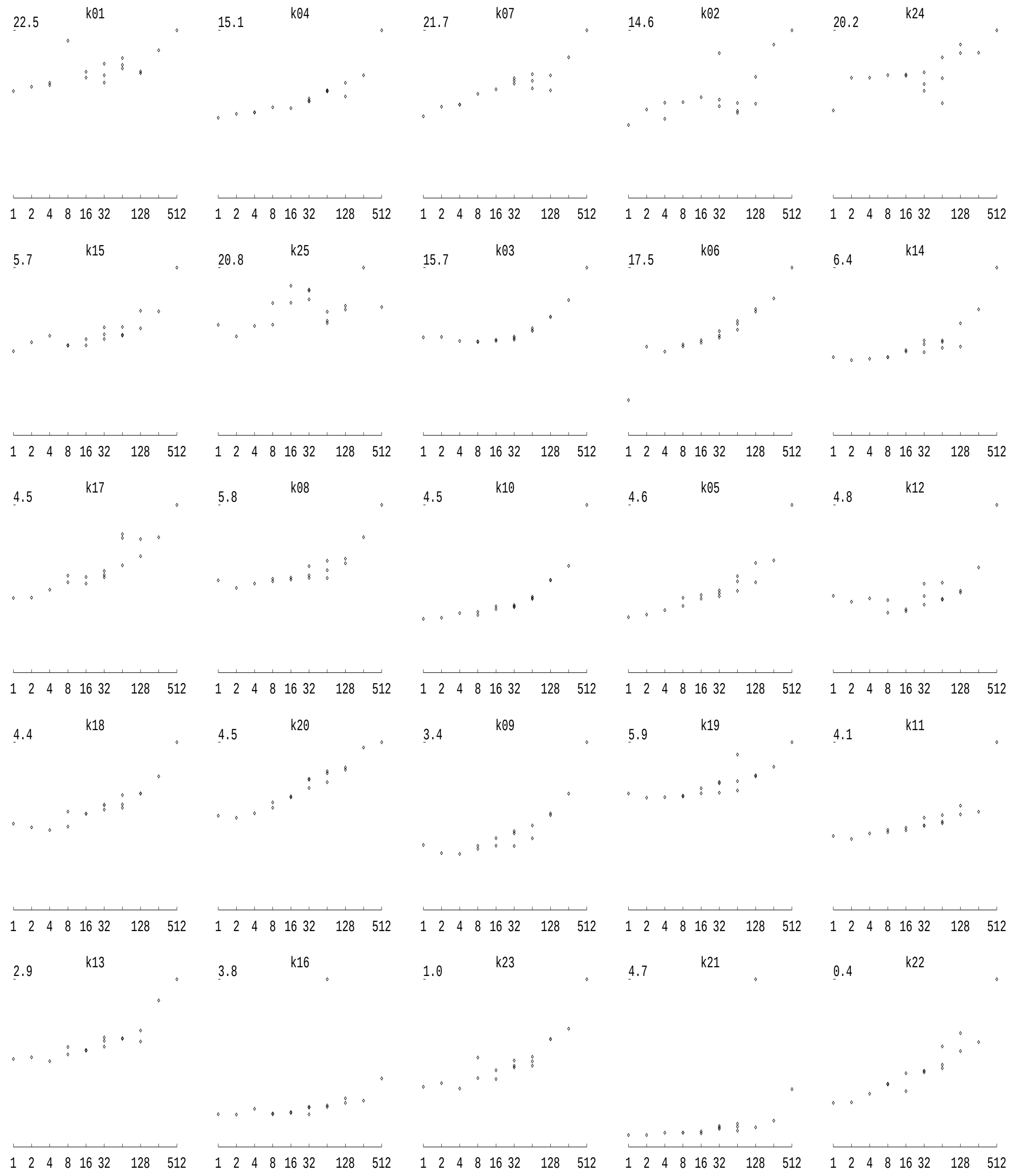

Figure 7: The critical path of each of Kaufman's 25 test positions, as measured on various machine sizes. Each box represents one test position. The positions are named `k01` through `k25`. The horizontal axis on each graph is the machine size. The vertical axis is the critical path length, in seconds. 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.

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 sitting idle to avoid swamping busy processors with requests for work.
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 8: 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 9: How the time is spent on ‘chess work’ for StarTech running on 512 processors on a typical problem.

A more complex scheduler 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 a more complex scheduler to handle context switching between subsearches on a single processor. These improvements are worth investigating.

It has been argued that using the Hitech static evaluator is a bad match for an all-software computer chess proram. 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 could be better than Hitech’s static evaluator, but given the overheads of StarTech’s search routines, simply speeding up the static evaluator would not make a huge performance difference. I believe that the main weakness of StarTech is its lack of search extensions rather than any weakness in the static evaluator.

The code for move generation and checking illegal moves, which takes a total of 3.5% of the cycles, was optimized by hand in assembly language. Before the optimization, the move generation and illegal move checking accounted for about 9% of all the cycles.

Three Improvement Strategies

I found three strategies to improve the performance of StarTech: Recursive iterative deepening, deferred-reads, and a slight serialization of the search.

The first strategy for improving performance is to perform recursive iterative deepening in order to improve move ordering. StarTech uses its global transposition table to improve move ordering. Most other programs use additional move-ordering mechanisms such as the killer table [GEC67] and the history table [MOS86]. StarTech does not use any such additional move-ordering heuristics. Recursive iterative deepening works as follows. 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. Recent experiments performed by D. Dailey on his Socrates program suggest that recursive iterative deepening may actually slow down programs that already have good move-ordering heuristics [Dai94]. (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].) Without recursive iterative deepening, StarTech chooses the right first move 85%–95% of the time. With recursive iterative deepening, a few percent more of the positions are searched in best-first order. Recursive iterative deepending is worth about a 20% performance improvement in StarTech.

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. Deferred-reads are worth about a 4% performance improvement in StarTech.

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 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 efficiency of StarTech by 10%–15% without substantially increasing the critical path length.

Serialization Heuristics for Jamboree Search

The story of that third strategy illustrates how critical path and work can be used to make good decisions about tuning a parallel chess program. During the development of StarTech I found several heuristics that might improve the efficiency of the Jamboree chess algorithm on real chess positions. This improvement in efficiency often came at the expense of an increased critical path length. I found one heuristic that actually improves the performance without significantly increasing the critical path, 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.

I arranged for StarTech to compute 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%–30% of all the work is cutoff work, while less than 2% is failed work.

I found that depending on the position, 50%–90% of the failed work is on Type 2 positions (positions that in a best-ordered tree are off the principal variation and immediately fail high) that dropped below α, while 10%–40% of the failed work is on Type 3 (positions that are off the principal variation and do not fail high) 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 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 not to serialize the positions completely 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%–15% while only increasing the critical path slightly, so that the average parallelism was still over 500.

By measuring average parallelism we can understand the impact of our algorithm design decisions. In contrast, one recent enhancement to the Zugzwang program [FMM93] 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. They express concern that by serially searching the first child before starting the other children they have reduced the average parallelism. Since the Zugzwang literature does not analyze critical path lengths, it is difficult to determine how Zugzwang's serialization scales with the machine size without actually running the program on a big machine. Measuring critical path and work can answer such questions.

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 misled 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.

5 Conclusions

Computer chess provides a good testbed for understanding multithreaded computations. The parallelism of the application derives from a dynamic expansion of a highly irregular game-tree. 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.

By separating the search algorithm and the scheduler, the problems of each could be solved separately. Once I had built a provably good scheduler, I was able to focus my attention on the application, analyzing and improving the performance of the underlying search algorithm. By using critical path to understand the a program, one can make good tradeoffs in algorithm design. Without such a methodology it can be very difficult to do algorithm design. My measurements demonstrate, for example, that there is plenty of parallelism in StarTech.

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. 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.

I found that chess places great demands on a scheduler. In another experiment I performed, I found, by reconstructing the schedule for an infinite-processor simulation, that sometimes there is plenty of parallel work to do, and sometimes there is very little. I typically saw average parallelism of at least several hundred, but for about a quarter of the run-time on an infinite processor machine, the average 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.

StarTech's tournament performance demonstrates the practicality of the parallel computer chess technology described in this paper. StarTech, 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 Computer Chess Tournament on its first outing.

The Future

The StarTech work points to several areas for future work, including new algorithms, new programs, and new programming paradigms.

There are several other approaches to game tree search that are not based on α-β search, several of which might be applicable to parallel search. For example, H. Berliner's B* search algorithm [Ber79] tries to prove that one of the moves is better with respect to a pessimistic evaluation than any of the other moves with respect to an 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. Nonetheless, smarter algorithms with higher overheads may become more valuable as machine performance increases.

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.

I have been working on a newer program, called $\star$Socrates, with D. Dailey, L. Kaufman, C. F. Joerg, C. E. Leiserson, R. D. Blumofe, M. Halbherr, and Y. Zhou [JK94]. $\star$Socrates uses more sophisticated search extensions and seems to have even greater average parallelism than StarTech. $\star$Socrates uses a new programming language and run-time system being developed at MIT called Cilk (pronounced 'Silk') [BJK*94]. Cilk provides a language and run-time system to separate the application program from the problems of scheduling and load balancing on a parallel computer. Cilk hopes to make it possible for ordinary C programmers to write multithreaded applications without having to be experts in parallel computing.

Acknowledgments

This work spans three universities and one corporation, with contributions from people at Carnegie-Mellon University (CMU), the Massachusetts Institute of Technology (MIT), the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign (NCSA), and Thinking Machines Corporation. The following people helped in various ways too numerous to recount: Robert D. Blumofe, Mark Bromley, Roger Frye, Richard Karp, John Mucci, Ryan Rifkin, James Schuyler, David Slate, Larry Smarr, Lewis Stiller, Kurt Thearling, Richard Title, Al Vezza, David Waltz, and Michael Welge.

Prof. Charles E. Leiserson supervised this research and was a continuous source of good ideas for StarTech.

This research would not have been possible without the help provided by Hans Berliner and Chris McConnell. Hans and Chris gave me a version of Hitech, and provided much helpful advice over the several years that I spent working on StarTech.

References

[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.

[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.

[Ber93] Hans Berliner. Personal communication. October 1993.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94),* Santa Fe, New Mexico, November 1994. (To appear.)

[BJK*94] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Andrew Shaw, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. December 1994. Submitted for publication. (Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/cilk/cilkpaper.ps.Z`.)

[Dai94] Don Dailey. Personal communication. June 1994.

[Elo78] Arpad E. Elo. *The Rating of Chessplayers — Past and Present.* Arco Publishers, New York, 1978.

[FMM93] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. In *Advances in Computer Chess 7,* 1993.

[Fis84] J. P. Fishburn. *Analysis of Speedup in Distributed Algorithms.* UMI Research Press, Ann Arbor, MI, 1984.

[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.

[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.

[JK94] Christopher F. Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Third DIMACS Parallel Implementation Challenge Workshop,* Rutgers University, October 1994. (Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/cilk/dimacs94.ps.Z`.)

[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.

[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.)

[Kus94] Bradley C. Kuszmaul. *Synchronized MIMD Computing.* Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1994. (Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/bradley/phd.ps.Z`.)

[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.

[Pea80] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence,* **14** (2), pages 113–138, September 1980.

[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.

[Sto79] G. C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, **12** (2), pages 179–196, August 1979.

[Tru92] Tom Truscott. Personal communication. September 23, 1992.