# Massively Parallel Chess

Christopher F. Joerg[1] and Bradley C. Kuszmaul[2]
{cfj,bradley}@lcs.mit.edu

MIT Laboratory for Computer Science
NE43-247, 545 Technology Square
Cambridge, MA 02139

(To be presented at the DIMACS'94 Challenge, October 1994)

## Abstract

Computer chess provides a good testbed for understanding dynamic MIMD-style computations. To investigate the programming issues, we engineered a parallel chess program called *Socrates, which running on the NCSA's 512 processor CM-5, tied for third in the 1994 ACM International Computer Chess Championship. *Socrates uses the Jamboree algorithm to search game trees in parallel and uses the Cilk 1.0 language and run-time system to express and to schedule the computation. In order to obtain good performance for chess, we use several mechanisms not directly provided by Cilk, such as aborting computations and directly accessing the active message layer to implement a global transposition table distributed across the processors. We found that we can use the critical path $C$ and the total work $W$ to predict the performance of our chess programs. Empirically *Socrates runs in time $T \approx 0.95C + 1.09W/P$ on $P$ processors. For best-ordered uniform trees of height $h$ and degree $d$ the average available parallelism in Jamboree search is $\Theta((d/2)^{h/2})$. *Socrates searching real chess trees under tournament time controls yields average available parallelism of over 1000.

## 1 Introduction

Computer chess provides a good testbed for understanding dynamic MIMD-style computations. The parallelism in computer chess is derived from a dynamic expansion of a highly irregular game-tree, which makes computer chess difficult to express, for example, as a data-parallel program. To investigate how to program this sort of dynamic MIMD-style application, we engineered a parallel chess program called *Socrates (pronounced "Star-Socrates".) The program, based on Heuristic Software's serial Socrates program, has an informally estimated rating of over 2400 USCF. *Socrates, running on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) at the University of Illinois, tied for third place in the 1994 ACM International Computer Chess Championship held at the end of June 1994 in Cape May, New Jersey.

*Socrates is a step forward from *Tech [Kus94], our previous chess program. *Tech is based on H. Berliner's serial Hitech program [BE89], and running on NCSA's 512-node CM-5, tied for third in the 1993 ACM International Chess Championship. *Socrates borrowed many of the techniques we developed for *Tech, including the basic search algorithm and the transposition table. *Socrates uses a new programming language and run-time system called Cilk 1.0 [BJK*94] to separate the chess program from the problems of scheduling and load balancing on a parallel computer.

To help manage the complexity of our chess systems, we divided the programming problem into two parts: an application and a scheduler. The application can be thought of as a dynamically unfolding directed acyclic graph, where the graph vertices correspond to instructions, and the graph edges correspond control-flow dependencies between various instructions. An instruction may not execute until all its predecessors have executed. The scheduler, on the other hand, takes such a DAG and decides on which processor each instruction should run, and when it should run. The application's job is to expose parallelism. The scheduler's job is to run the program as fast as possible, given the available parallelism in the application, without running out of memory. Thus, in *Socrates, we use Cilk 1.0 to address the scheduling problem, and the chess program itself can focus on only those issues which are unique to a chess program.

We had learned from our previous parallel chess program, *Tech, how to predict the performance of a parallel chess program. It was not clear from the outset 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 available parallelism. Chess programs use large global data structures and are nondeterministic. We wanted predictable performance. For example, if one develops a program on a small machine, one 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? We had found from *Tech that there are two complexity measures of performance that actually can predict the 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.

[1]http://csg-www.lcs.mit.edu:8001/Users/cfj/
[2]http://theory.lcs.mit.edu/~bradley

of chess programs: the total work $W$ and the critical path length $C$.

The total work and critical path length give us a chance to understand how the performance of a parallel program will scale as the number of processors increase, and also gives us a chance to understand the effectiveness of our scheduler. For example, the effectiveness with which the available work is scheduled into the machine can be measured by comparing it to the bound from Brent's theorem [Bre74, Lemma 2], which states that the runtime on $P$ processors with a perfect scheduler can be brought down to no more than $C + W/P$.

The values of $W$ and $C$ depend on the parallel algorithm, rather than on the scheduler. In our game-tree search algorithm, the values of $W$ and $C$ are partially dependent on scheduling decisions made by the scheduler, but we believe that $W$ and $C$ are mostly independent of those decisions. A good algorithm reduces $W$ and $C$. We can compare $W$ to the runtime of a corresponding serial chess program, and we can compare $C$ to $W$. The ratio of $W$ to the work done by the serial program is the *efficiency* of the program, and indicates how much overhead is inherent in the parallel algorithm. The ratio $W/C$ is the average available parallelism of the program. We can hope, because of Brent's theorem, to use as many as $W/C$ processors with an efficiency of at least 50%.

This paper explains how we obtain predictable high-performance on *Socrates. Section 2 describes the Jamboree game-tree search algorithm and presents some analytical results describing the performance of Jamboree search. Then, in Section 3, we describe the Cilk 1.0 language and run-time system. The modifications made to Cilk in order to run the chess program are described in Section 4. In Section 5 we outline several other mechanisms used in the chess program. Section 6 presents a description of how the Jamboree algorithm relates to the algorithms used by other chess programs. We make some concluding remarks in Section 7.

# 2 Parallel Game Tree Search

The *Socrates chess program uses an efficient parallel game-tree search algorithm called "Jamboree" search. In this section we explain Jamboree search, starting with the basics of negamax search and serial $\alpha$-$\beta$ search, and present some analytical performance results for the algorithm.

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.
- Each child that turns out to be better than the first child is searched in turn 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.

## 2.1 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. (Readers who are familiar with the serial game tree search algorithms may wish to skip directly ahead to the description of the Jamboree algorithm in Section 2.4.) 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:

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

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 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.

## 2.2 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

```
(N1)  Define negamax($p$) as
(N2)      If $n$ is a leaf then return static_eval($n$).
(N3)      Let  $\vec{c} \leftarrow$ the children of $n$, and
(N4)          $b \leftarrow -\infty$.
(N5)          For $i$ from 0 below $|\vec{c}|$ do:
(N6)              Let $s \leftarrow -$negamax($\vec{c}_i$).                ;; Recursive Search
(N7)                  if $s > b$ then set $b \leftarrow s$.               ;; New best score
(N8)              enddo
(N9)          return $b$.
```

Figure 1: Algorithm negamax.

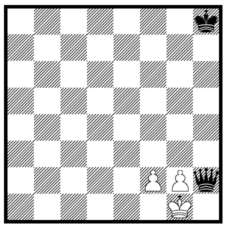


Figure 2: White to move and win. In this position, White need not consider all of Black's alternatives to 40. ♔f1, since almost any move Black makes will keep the queen, a worse outcome than just taking the queen with 40. ♔×h2.

the $\alpha$-$\beta$ 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 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 $\alpha$-$\beta$ 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 $\alpha$ and $\beta$. If the value of any child, when negated, is as great as $\beta$, then the value of the parent is no less than $\beta$, and we say that the parent *fails high*. If the values of all of the children, when negated, are less than or equal to $\alpha$, then the value of the parent is no greater than $\alpha$, and we say that the parent *fails low*.

Procedure absearch[3] is shown in Figure 3. When Procedure absearch is called, the parameters $\alpha$ and $\beta$ are chosen so that if the value of a node is not greater than $\alpha$ and less than $\beta$, 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 $\beta$. If so, we know that the value of the node is at least as great as $\beta$ and we can skip searching the remaining children; the node has failed high. Just because one of the children has a negated score less than $\alpha$, however, does not mean that some other child might not be within the $\alpha$-$\beta$ window. The algorithm can only fail low after considering all of the children.

The $\alpha$-$\beta$ algorithm can substantially reduce the size of the tree searched. The $\alpha$-$\beta$ 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 $\alpha$-$\beta$ 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 $\alpha$-$\beta$ algorithm obtains the value of the $k$th child and possibly uses that value to adjust $\alpha$ or return immediately. This dependency between finishing the $k$th child and starting the $(k+1)$st child completely serializes the $\alpha$-$\beta$ search algorithm.[4]

## 2.3 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 $\alpha$-$\beta$ search, called *Scout* search, and modified it to be a parallel algorithm. This section explains the Scout search algorithm.

[3]This variant on the standard $\alpha$-$\beta$ algorithm is apparently due to Fishburn [Fis83], who called it *fail-soft $\alpha$-$\beta$ search.* Fail-soft $\alpha$-$\beta$ search can return a value that is less than $\alpha$, 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 $\beta$, in which case the value returned is a lower bound to the true value.

[4]R. Finkel and J. Fishburn showed that if the serialization implied by $\alpha$-$\beta$ pruning is ignored by a parallel program, then it will achieve only $\sqrt{P}$ speedup on $P$ processors [FF82].

```
(A1)   Define absearch$(n,\alpha,\beta)$ as
(A2)       If $n$ is a leaf then return static_eval$(n)$.
(A3)       Let $\vec{c} \leftarrow$ the children of $n$, and
(A4)           $b \leftarrow -\infty$.
(A5)           For $i$ from 0 below $|\vec{c}|$ do:
(A6)               Let $s \leftarrow -$absearch$(\vec{c}_i, -\beta, -\alpha)$.
(A7)                   If $s \geq \beta$ then return $s$.                ;; Fail High
(A8)                   If $s > \alpha$ then set $\alpha \leftarrow s$.        ;; Raise $\alpha$
(A9)                   If $s > b$ then set $b \leftarrow s$.
(A10)            enddo
(A11)          return $b$.
```

Figure 3: Algorithm absearch.

```
(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$(c_0, -\beta, -\alpha)$.
(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 4: Algorithm scout.

Figure 4 shows the serial Scout search algorithm, which is due to J. Pearl [Pea80]. 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 $(-\alpha)$ 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 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.

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 score against $\beta$ on Line S11.

A test is typically cheaper to execute than a valuation because the $\alpha$-$\beta$ 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

```
(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 5: Algorithm `jamboree`.

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.

## 2.4 Jamboree Search

The Jamboree algorithm, shown in Figure 5, 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. 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.

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.

The actual search algorithm used in *Socrates also includes some *forward pruning* heuristics that prune a deep search based on a shallow preliminary search. The idea is that if the shallow search looks really bad, then most of the time a deep search will not change the outcome. Forward pruning techniques have lately been shown to be extremely powerful, allowing programs running on single processors to beat some of the best humans at chess. The serial Socrates program uses such a scheme, and so does *Socrates. In the *Socrates version of Jamboree search, we first perform the preliminary search, then we search the first child, then we test the remaining children in parallel, and research the failed tests serially.

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.

## 2.5 Analysis of Jamboree Search

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

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 following hold:*

- *The total work performed is $\Theta(d^{h/2})$, which is the same as serial $\alpha$-$\beta$ search would perform. That is, the work efficiency is 1.*
- *The critical path length is $\Theta(2^{h/2})$, and thus the average available parallelism is $\Theta((d/2)^{h/2})$.*

Chess trees typically have degree of between 30 and 40 in the middle-game, and since we hope to search at least 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 following hold:*

- *The total work performed is $\Theta(d^h)$.*
- *The critical path is $\Theta(d^h)$.*

*For large $d$ and $h$, the constants work out so that the total work performed is approximately three times as much as the serial $\alpha$-$\beta$ search would perform (thus the efficiency is $1/3$), and the critical path length is equal to the work performed by serial $\alpha$-$\beta$ (with the speedup approaching 1 from below.)*

Surprisingly, for worst-ordered uniform game trees, the speedup of Jamboree search over serial $\alpha$-$\beta$ search turns out to be under 1. That is, Jamboree search is worse than serial $\alpha$-$\beta$ search, even on a 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 not-quite linear speedup on worst-ordered trees [Fis84].

## 2.6 Real Chess Trees

For real chess trees, we found that the better the move ordering, the lower the critical path and the less total work is performed. Thus, the move ordering heuristics of a chess program, which are important for serial programs because it reduces the work, are doubly important for our parallel algorithm because it also decreases the critical path length.

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. Nonetheless, we have found that we can use the measurd critical path length and total work to tune the program.

Our strategy is to measure the critical path and the work on a particular run, and to try to predict the performance from those measurements. (The details of how we measure critical path length are discussed in Section 3.) We measured the program on a set of eight problems[5], shown in Figure 6. For each problem the program was run to various depths up to those that allowed the program to solve the problem by getting the "correct" answer, as identified by Kaufman. We also measured the programming running on a variety of different sized machines. Then we performed a curve-fit of the data to a performance model of the form

$$T_{\text{predicted}} = c_1 \cdot C + c_2 \cdot \frac{W}{P} + c_3.$$

We found that the performance can be accurately modeled as

$$T \approx (0.95 \pm 0.04)C + (1.091 \pm 0.001)\frac{W}{P} + 0 \qquad (1)$$

with a sample correlation coefficient[6] of 0.999947, and a mean error of 14.2% and a mean relative error of 4.85%.[7] To us, this is quite amazing, because chess is a very demanding application. For *Tech, we found that according to measurements of the ideal parallelism profile (which shows the amount of parallelism as a function of time, running the program on an ideal infinite processor machine), for half the run-time there is often less than 10-fold parallelism. The low coefficients on Equation 1 indicate that the program quickly finishes the available work during the

[5]Our eight problems were provided by *Socrates team member L. Kaufman, who is an International Master. Kaufman has published several larger sets of benchmarks [Kau92, Kau93] that were used to understand *Tech [Kus94].

[6]For a definition of sample correlation coefficients and other statistical terms see, for example, [HL93, page 51].

[7]The results presented here are for *Socrates. A more complete analysis of the statistical properties of the measurements for *Tech can be found in Kuszmaul's dissertation [Kus94].

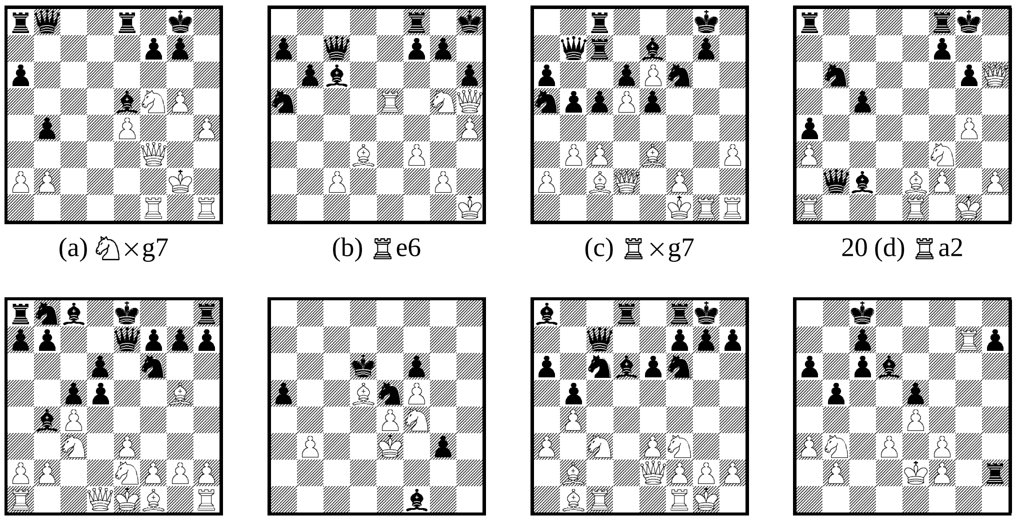


Figure 6: The 8 chess positions used in this paper. Below each position is shown Kaufman's "correct" move for that position. All positions are "White to move", except for Position (f).

times of low parallelism, and when there is much parallelism the program efficiently load balances the work.

We also found that the work increases by about a factor of two to three as the number of processors increases from 1 to 128 processors, and that the critical path length is fairly stable as the number of processors increases. Most of the difficulty of predicting the performance of the chess program comes from the fact that the amount of work is increasing. The processors end up expanding subtrees that are pruned in the serial code.

We found that the critical path does not limit the speedup for our test problems, or for the program running under tournament conditions. By using critical path to understand the parallelism of our algorithm, we are able to make good tradeoffs in our algorithm design. Without such a methodology it can be very difficult to do algorithm design. For example, Feldmann, Monien, and Mysliwietz find themselves changing their Zugzwang chess program to increase the parallelism without really having a good way to measure their changes [FMM93]. They express concern that by serially searching the first child before starting the other children they have reduced the available parallelism. Our technique allows us to state that there is sufficient parallelism to keep thousands of processors busy without changing the algorithm. We can conclude that we should try to reduce the total amount of work done by the program, even if it reduces the available parallelism slightly.

We experimented with some techniques to improve the work efficiency, and found several techniques to improve the work efficiency at the expense of increasing the critical path length. For example, on *Tech we considered a algorithm change that would value the first two children before starting the parallel tests of all the remaining children. The idea is that by valuing more children, it becomes more likely that the best of the children that have been valued will be able to prune some of the remaining children. When we measured the runtime on a small machine, the program ran faster but on a big machine the runtime actually got worse. To understand why, we looked at the work and critical path length. We found that this variant of Jamboree search actually does decrease the total work, but it increases the critical path length, so that there is not enough available parallelism to keep a big machine busy. By looking at both the critical path length and the total work we were able to extrapolate the performance on the big machine from the performance on the little machine, however, and so we avoided introducing modifications that would hurt us in tournament conditions.

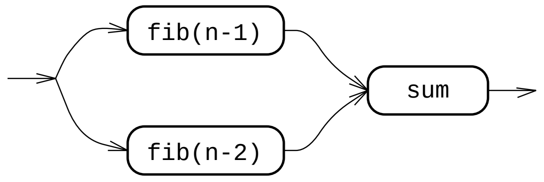
## 3 The Cilk Work-Stealing Scheduler

Now that we have explained the search algorithm used in *Socrates, we need to explain how the computation is distributed across the machine. We use a run-time system called Cilk 1.0 [BJK*94][8] to distribute work among the CM-5 processors. This section explains how a program is expressed in Cilk and how the computation is distributed across the machine.

To distribute work among CM-5 processors, Cilk uses a randomized work-stealing approach, in which idle processors request work. Processors run code that is nearly

[8]Cilk is a threaded language of the C ilk.

```
int fib (int n)
{
        if(n<2) return n;
        else return fib(n-1)+fib(n-2);
}
```



```
thread sum (cont k, int x, int y)
{
        SendWordArgument (k, x+y);
}

thread fib (cont k, int n)
{
        if (n<2) SendWordArgument (k, n);
        else
        {       cont x, y;
                spawn_next sum (k, ?x, ?y);
                spawn fib (x, n-1);
                spawn fib (y, n-2);
        }
}
```

Figure 7: Expressing the doubly recursive Fibonacci program in Cilk 1.0. On the upper-left is the program written in serial C. On the lower-left is the dataflow graph for the program. On the right is the corresponding Cilk code.

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 work locally, it sends a message to another processor, selected at random, and removes work from that processor's collection of posted work.

The Cilk system was original based on the Parallel Continuation Machine run-time system of Halbherr, Zhou and Joerg [HZJ94]. In PCM, the scheduler uses a double ended queue (a *deque*) on every processor. When a processor posts work, it pushes it on the bottom of the deque. When a processor needs more work to do locally, it pops it off the bottom of the deque. When a processor steals work, the work is stolen from the top of the deque on the remote processor. It turns out that we modified this basic scheduler, as we shall describe in Section 4.4.

Cilk requires that the programmer explicitly break the algorithm into threads. To give an idea of how programs are expressed, consider the doubly recursive Fibonacci program shown in Figure 7. First we convert the program to a dataflow graph, and then for each node of the graph, we write a thread, which looks like a C function. Thus, in the final Cilk code, there are two threads, the `sum` thread and the `fib` thread. The `sum` thread accepts two values, adds them, and sends the result to an explicitly provided continuation. The `fib` thread creates a thread to sum two results, and passes continuations (denoted `x` and `y`) for that thread to two subsidiary `fib` threads. For a more complete description of the Cilk syntax, including a tutorial, see [BJK*94].

Similarly for the Jamboree algorithm, we transform the search code shown in Figure 5 into a dataflow graph, as shown in Figure 8. Then we express the program in Cilk analogously to the Fibonacci example.

Cilk automatically computes the critical path length and total work of a computation. The computation of the critical path is done by a system of time-stamping, as shown in Figure 9.

The Cilk system runs on both the CM-5 and network of workstations. Soon we expect to provide Cilk versions that run on shared memory multiprocessors and a variety of other parallel computing platforms. We are currently working on improving the Cilk time system to provide better support for global data structures, for input/output, and to help automatically break up a program into threads.

## 4 Using Cilk for Chess Search

In the following two sections we describe the implementation of *Socrates using Cilk. These sections are an interesting case study in implementing a large, multithreaded, speculative application. As mentioned in the introduction, *Socrates is a parallelization of a serial chess program. Much of the code, including the static evaluator, is identical in the parallel and the serial versions and is not be discussed here. Instead, we focus on the portions of the code which were written specifically for the parallel version.

This section focuses on those parts of the Cilk scheduler that we had to change in order to make Cilk behave more like the scheduler used in *Tech. The changes we made include implementing migration handlers, aborting computations that are in progress, changing the order in which threads are stolen, and adding level waiting.

### 4.1 Migration Threads

We use a large, variable sized data structure (nearly 200 bytes) to describe the state of a chess board. In the serial

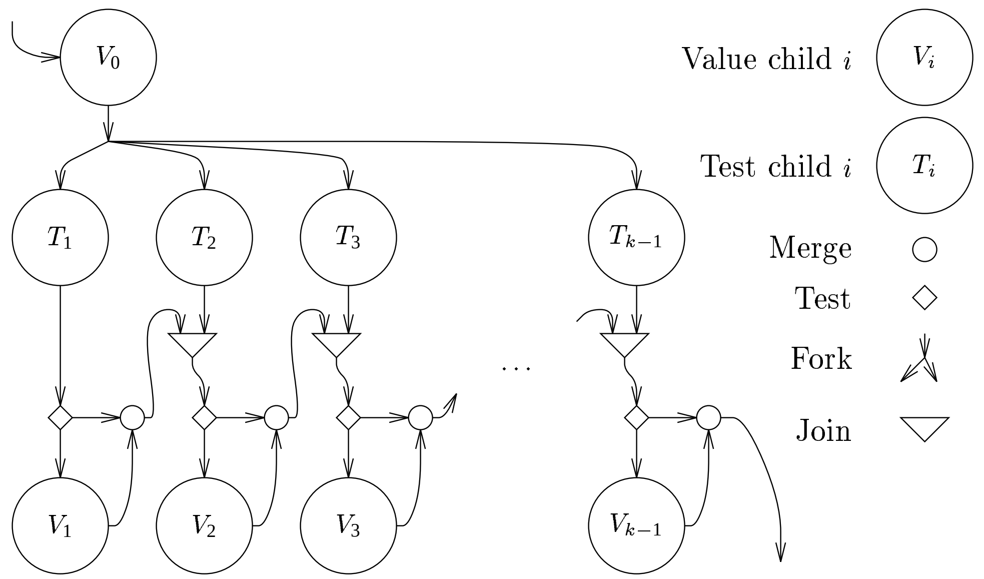


Figure 8: The dataflow graph for Jamboree search. First Child 0 is searched to determine its value, then the rest of the children are tested in parallel to try to prove that they are worse choices than Child 0, and then each of the children that fail their respective tests are serially researched. This 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 5.

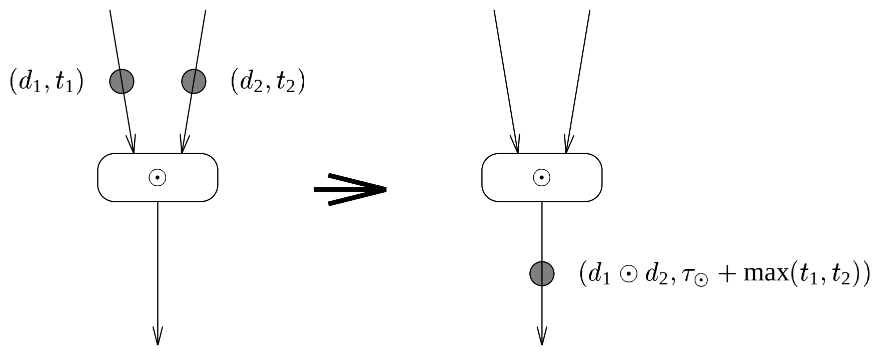


Figure 9: The time at which an instruction in a dataflow graph is executed in a perfect infinite-processor schedule can be computed by time-stamping the tokens. In addition to the normal data-value of a token ($d_1$, $d_2$, and $d_1 \odot d_2$ respectively in the figure), the token includes a time-stamp ($t_1$, $t_2$, and $\tau_\odot + \max(t_1, t_2)$ respectively.) The time-stamp on the outgoing token is computed as a function of the time-stamps of the incoming tokens and the time to execute the instruction.

code we pass around pointers to this structure and copy it only when necessary. In the parallel code we cannot just blindly pass pointers between threads, because if the thread is migrated the pointer will no longer be valid. A naive solution is to copy the state structure into every thread, but this adds a significant overhead to the parallel code. This overhead is especially distasteful when you realize that well under 1% of threads are actually migrated, so most of the copying would be wasted effort.

To solve this problem we use migration threads. Any thread can have a migration thread associated with it. When the scheduler tries to migrate a thread that has an associated migration thread, the scheduler will first call the migration thread. This migration thread will return a new closure which is migrated instead.

Using this mechanism we are able to pass threads a pointer to a state structures. Any thread that is passed a state pointer is also given a migration thread which will copy the state into the closure if the thread is stolen. Once the closure arrives at the stealing processor, the stolen thread can then be called with a pointer to the copied state structure. This allows the overhead of copying the state to be paid only when it is actually necessary.

## 4.2 Abort

In order to implement the jamboree search algorithm we must be able to abort a computation. This is needed when we discover that at least one child has a score greater than beta, so there is no need to search the rest of the children. (This is called *failing high*.) The Cilk system has no built-in mechanism for aborting a computation, so this had to be added as user code. Our goal in designing the abort mechanism was to keep it as self contained as possible and to minimize changes to the rest of the code. Eventually we would like to add support for such a mechanism to Cilk itself.

In order to abort a computation we must be first able to find all of the threads that are working on this computation. To implement this we use *abort tables* to link together all the threads working on a computation. When a computation, say $A_1$, needs to create several children it first creates an abort table containing an entry for each child of the computation. If a child of $A_1$, say $B_1$, itself spawns off children, then the entry for $B_1$ is updated to contain a pointer to the abort table that $B_1$ creates. Once $B_1$ and all its children have completed, $B_1$'s table is deallocated and the entry for $B_1$ is updated. With this mechanism in place the abort code is able to find all the descendants of any computation. When performing an abort, the abort code does not actually destroy any threads, instead it merely makes a mark in the affected abort tables. When a user's thread runs its first action should be to check to see if it has been aborted, and if so skip the rest of its computation. This check allows the user's code to do any cleaning up that may be necessary. (For example, the code may need to free some data structures.)

The abort mechanism provides functions to create, update, and deallocate the abort structures; to check if a thread is aborted; and to start an abort. By using these functions and passing around a few pointers to abort tables, the search code was modified to include aborting without too many changes.

One difficulty encountered in implementing the abort tables was in keeping the tables correct when a computation migrates. When a computation is stolen an abort table is allocated on the stealer's side and the existing abort table is modified to point to it. The difficulty arises because at the time a computation is stolen there is not yet an abort table on the stealer's side to point to. This abort table is not be allocated until after the thread begins to run (unless we change the run time system, which we wanted to avoid). So instead we create a unique identifier (UID) for each stolen computation, and store that into the abort table. Then on the stealer's side we have a hash table to map the UID into a pointer to the abort table. The protocol for accessing the hash table is quite tricky since there are many cases which require special handling. For example, the network of the CM-5 can reorder messages, therefore we have to handle the case where a message to abort a computation arrives before the thread that will allocate the hash table entry and abort table for that computation. Unfortunately, we did not consider all such possibilities before beginning the design, so getting this mechanism working correctly took longer than anticipated.

## 4.3 Steal Ordering

In the original Cilk runtime system the thread queue consisted of a single double ended queue. Newly enabled threads were placed at the front of the queue and the local processor took work out of this side as well (i.e. LIFO). When stealing occurs, threads are stolen from the other side of the queue (i.e. FIFO). For a tree shaped computation, the LIFO scheduling allows the computation to proceed locally in a depth first ordering, thus giving us the same execution order a sequential program would have. However when stealing occurs the FIFO steal ordering causes a thread near the top of the tree to be stolen, so a large piece of work will be migrated, thus minimizing stealing. Since jamboree search is a tree shaped computation this mechanism works reasonably well.

With this scheduling mechanism, the order in which children are executed depends on whether or not a child is stolen. For most computations this execution order does not matter; but for jamboree search it does. Execution order has an effect because if one child fails high, the rest of the children do not need to be searched. Our program orders the children such that when no children are stolen (the common case) the children most likely to fail high are executed first; this order minimizes the total work $W$. The problem is that when stealing occurs we steal the child least likely to fail high.

Ideally we would like to steal from the top of the tree, but still steal the child that is most likely to fail high. To do this we had to modify the scheduler by adding the concept of levels. Each thread in the queue is assigned a level and threads at the same level will be executed in a fixed order, regardless of whether they are stolen or executed locally. Between levels, however, scheduling is done as before: We execute locally at the shallowest (newest) level and steal from the deepest (oldest) level. The search code then marks all the children of a computation as being at a level one shallower than the level at which the computation is currently executing. This gives us exactly the ordering of threads that we want. Adding this to *Socrates reduced the amount of work performed for searching a position and seemed to give a speedup of 20-25%. This idea seemed important enough that we included a cleaner version of this mechanism in Cilk 1.0.

### 4.4 Level Waiting

The final change we made to the scheduler was a further attempt to reduce the extra work being performed by the parallel version. When a processor is searching a board position, $A$, it spawns off a bunch of children to test. If a processor ran out of children to work on while some children were still being worked on elsewhere, that processor would steal another closure and begin working on that.

Consider the case where one (or more) of the children is stolen and the processor finishes the rest of the tests before the test of the stolen child completes. The processor may then be out of work to do[9]. This processor will then steal some closure from another processor and begin searching its board position, call it $B$. Eventually the test of the stolen child will complete. When this result comes back it will restart the computation on position $A$ and preempt $B$. Since position $A$ may still have additional value searches to perform, this is potentially a long computation. We are now in a position where $B$, no matter how little work it has, will not complete until the potentially long computation for $A$ completes. The computation which spawned $B$ will continue without it. It may eventually block (and thereby artificially lengthen the critical path $C$) or it may be able to continue, but will use looser bounds than if $B$ had completed (and will thereby increase the total work $W$).

To avoid this stalled work we further modified the scheduler. We added "level waiting", a feature which makes uses of the same levels that were used in the previous section for optimizing the steal ordering. When a computation spawns children all the sub-computations are placed at the same level. The level waiting mechanism simply requires that all of these sub-computations have completed before we may begin any work at a shallower level. This prevents us from starting, and then preempting, an unrelated search. Implementing this change seemed to give us a 15-20% speedup.

[9]It will often be out of work because none of the children at this level would have been stolen if there were any work earlier in the queue.

## 5 Other Chess Mechanisms

The previous section described issues that arose in getting the search routines to run in our parallel environment. This section describes other aspects of the serial code that had to be modified to run in a parallel system. These aspects include the transposition table, detecting repeated moves, and debugging support.

### 5.1 Transposition Table

Most serial chess programs include a Transposition Table. This is basically a hash table of previously evaluated nodes. After a node is searched we create (or update) the hash entry for this node. The information stored in this entry includes a score, a move, a depth and a check key. The score tells us the value of the node; the move tells us what move achieves this score; and the depth tells us how deep a search was done. The check key is used to distinguish between the many positions which may hash to this entry.

Before searching a node we first check to see if it is present with a deep enough depth, then we need not search this node again. This can occur because the same position can be reached by many different sequences of moves (i.e. a transposition). Much of the time when we get a hit the depth is not sufficient for the current search. But even in this case the table is still useful because it gives us the best move found by an earlier search, and often the best move at a shallower depth is the best move at a deeper depth. By using the returned move as our predicted best move, we increase our chances of accurately predicting the best move, which, as we saw in Section 2, reduces the work and critical path of the computation.

For *Socrates we implemented a distributed transposition table. We had a choice between implementing a blocking or a non-blocking interface to the table. When a thread begins a search of a node the first thing it typically does is to do a transposition table lookup on that node. In a blocking implementation, this thread would send off a lookup request to the appropriate node and busy-wait until the response arrives, and then continue. The obvious disadvantage of blocking is that we waste time busy-waiting.

In a non-blocking implementation we would break this thread into several threads. When the time came to do a lookup, a thread would be posted on the node that would hold the entry. This thread would do the lookup and send the result back to the original node, enabling the continuation of the search. This implementation has the advantage that we do not spend any time busy-waiting while we do a table lookup. But it has one big disadvantage in that it may lead to many searches taking place on the same node concurrently. Intermixing two or more searches on the same node can cause both the work and the critical path to increase. To avoid these increases the scheduler would have to be modified to keep the two computations separate. To avoid the complexity involved in such a modification we chose to implement a blocking transposition table.

Since there is no way to implement this blocking mechanism using Cilk primitives, we dropped to a lower level and used the Strata active message library [BB94]. We designed the transposition table such that all accesses are atomic. For example when a value is to be put into the table, the information about the position is sent to the node where the entry resides, and that node updates the entry as required. Alternatively, we could have implemented a non-atomic update by performing a remote read of the entry, modifying the entry, and then doing a remote write. Non-atomic updates would have required more messages and would have had to either lock the entry while the update was in progress, or risk losing some information if two update operations overlapped.

To determine how much the busy-waiting hurts us, we instrumented our code to measure the time spent busy-waiting[10]. Our experiments showed us that the mean time between sending the request and receiving the reply was around 1600 cycles. This worked out to about 7% of the execution time.

Another decision we faced was how large to make the hash entries. Clearly, we would like to make them as large as possible[11]. The score and the move each require 16 bits. The bits describing the depth and type of search required another 9. The only other piece of an entry is the check bits. In our implementation each position had a 64 bit key. Of these bits 9 were used to select a processor and 21 were used to select a hash line on a given processor, so there is no need to store these bits in the entry itself. Of the remaining bits 34 bits we stored only 23 of them as the check bits since this allowed us to fit an entry in one 64 bit double word. When executing on the 512 processor system we had a 1 billion entry hash table!

The last aspect of the transposition table we will examine is subsumptions. The issue is what, if anything, do we do if two independent searches are concurrently searching the same position (i.e. one search "subsumes" the other). For example, Processor P1 may begin a search of Position $B$ and before it completes and writes its result into the hash table Processor P2 begins another search of Position $B$. This leads to part of the search being duplicated. In the serial code these searches would be performed sequentially so this problem would not occur.

We considered trying to avoid this overhead in the following manner. When a search begins if the transposition table lookup fails an entry is created for that position and it is marked as "search in progress." Then if another lookup occurs on this position we know that a search is already being done. We would then have the option of waiting for the earlier search to complete.

We chose not to implement this mechanism. Implementing it would have been somewhat complicated, and there were a number of issues that this would raise that we did not have a clear understanding of. For example, when we were about to abort a search would it be necessary to first check to see if anyone else is waiting for the results of this search. Another example is deciding when to wait: If a position is already being searched to depth $d$, and we want to search it to depth $d-1$, do we wait for the deeper search? If we don't wait we are doing extra work, if we do wait we may wait much longer than if we had just done it ourself. We instrumented our program to estimate how much duplicate work was being done. Each time we completed a search and were about to write the hash table entry we first did a hash table lookup to see if we would get a hit if we began the search now. (If so, then someone else must have completed a search of this node during the time since we began the search.) We found that this occured less than 1% of the time. Furthermore, we had implemented a similar mechanism for *Tech, and it sometimes speeds the program up, and sometimes slows it down.

## 5.2 Repeated Moves

To fully describe a position in a chess game we need more than just a description of where each piece is on the board; some history is needed as well. A simple example is we need to know if the king has moved. If it has then we cannot castle, even if the king has moved back to its original position. This sort of information can easily be stored in a few bits in the state so this causes no difficulty.

Other required history can not be stored so easily. In chess if the same position is repeated 3 times then the game is a draw. Similarly if 50 moves are made by each player without an irreversible move being made, the game is a draw[12]. To handle these cases we need to keep track of all moves since the last irreversible move. (Once an irreversible move is made earlier positions cannot be repeated.) We do this by adding an array of positions to our state structure. This array contains all the positions (repre-

[10]Not all this time is wasted since while busy-waiting we poll the network so we may spent part of this time responding to arriving messages. But the analysis above gives us an upper bound on the cost of busy-waiting.

[11]Hsu claims that increasing the size of the hash table by a factor of 256 can easily give a factor of 2 to 5 speedup [Hsu90].

[12]An irreversible move is one which cannot be undone; that is, one which captures a piece or moves a pawn.

sented by their 64 bit hash key) since the last irreversible move.

This array greatly increases the size of the state structure (from about 160 bytes to nearly 1000 bytes). For a serial program the size of the state may not be significant since the code could just modify and unmodify the same state structure. For parallel code, however, it is often necessary to make copies of the state so a large state can slow down the program. To prevent this from occuring when we copy a state we only copy the part of the repeated position array that is meaningful. Since the average length of this list is quite small (under 2) copying this list adds very little overhead.

### 5.3 Debugging

In order to make it easier to debug our code, we make liberal use of 'assert' statements. Not only did this cause bugs to be detected sooner, it was also helpful in pinpointing the cause of the bug. One of our biggest problems initially was making sure that the parallel version was working correctly. This was difficult because if the parallel version was close to the serial, but not exactly the same, it would usually produce the exact same answers. We were often modifying both the parallel and the serial search algorithms and keeping them consistent was quite error prone. One method we occasionally used to test whether both versions were identical was to run the parallel code on one processor and run the serial code and make sure they both searched exactly the same number of nodes. Unfortunately we did not do this check often enough and at one point so many minor variations had crept in that we wound up spending almost a week trying to make both versions consistent again.

One of the most useful assertions we added was to check at every node of the tree that the results of the parallel code were the same as the serial code. In the debugging version of the code, after the search of a position was complete we would call the serial code on the same position and assert that the results were the same. (We do this with the hash table turned off, otherwise the serial code simply finds the result in the hash table.) This was extremely slow, but it is an easy way to detect any differences between the serial and parallel searches, and to pinpoint exactly where the differences lie. After we started using this check, keeping both versions identical became much easier. We think this is an approach that is applicable to many parallel programs, not just chess.

Even with this grandiose verification not all our bugs were detected. At one point the debugging mode worked fine when run on any number of processors, as did the non-debugging program when run on one processor. But when we ran on more than one processor the speedup was quite small. It turned out that debugging mode was not being completely turned off as the flag which says whether or not to use the hash table was being set correctly only on processor 0. Therefore all other processors would never use the hash table. As is often the case, bugs which affect only performance can be harder to detect than bugs that affect correctness.

## 6 Related Search Algorithms

Our chess program uses *Jamboree* search [Kus94], a parallelization of scout search [Pea80], 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 it has 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]. We do not claim that the search algorithm is a new contribution. Instead, we view the algorithm as a testbed for evaluating mechanisms needed for the design of scalable, predictable, asynchronous parallel programs.

Jamboree search was used in our previous program, *Tech [Kus94]. *Socrates is a step forward compared to *Tech because we introduced a linguistic layer and runtime system called Cilk 1.0 [BJK*94] to make it easier to program the application without worrying about the scheduling issues. Many of the techniques originally used in *Tech were borrowed for *Socrates. Inspired by some problems we had with early versions of our *Tech program, Leiserson and Blumofe designed a provably good scheduler that has good space and time bounds, as well as low communications requirements [BL94].

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 $\alpha$-$\beta$ search by searching the minimal tree (i.e., the Knuth-Moore critical tree [KM75]). Each position is kept in an expanded form, potentially for 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 $\alpha$-$\beta$ 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 $\alpha$-$\beta$ 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. In contrast, Jamboree researches sequentially. For nodes with exactly one critical child, MWF searches just 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 $\alpha$-$\beta$ 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. Our Theorem 2, in contrast, states that for an infinite processor perfect scheduler the average available parallelism is less than 3 and the speedup is less than one. Even though the MWF algorithm is tangled up with the tree-of-processors scheduler, one can interpret Fishburn's results somewhat 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 serially 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 Cray Blitz, however.

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 our programs 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 $\alpha$-$\beta$ 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 *Tech, use a serial aspiration search in which the game tree is searched with a small $\alpha$-$\beta$ 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 $\alpha$-$\beta$ 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 $\alpha$-$\beta$ search. 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. We wanted to be able to compare our work to the best serial algorithms.

## 7 Conclusions

The history of *Socrates sheds some light on the problems of developing a high-performance parallel program. The *Socrates chess team, which includes includes R. Blumofe, M. Halbherr, C. Joerg, B. Kuszmaul, C. Leiserson, and Y. Zhou of MIT as well as D. Dailey and L. Kaufman of Heuristic Software, decided to start with a new chess program rather than to try to parallelize the original Socrates program. The difficulty with the original Socrates program is that it uses many global variables which are modified throughout the search. We felt that it would be easier to start with a program that was designed to modify its state in a non-destructive fashion by always making a new copy of the variables that represent the state of a chess board in the tree search. It turned out that the decision to start with a new program resulted in the program being substantially weaker than we had hoped, because we did not have sufficient time to get all of the chess knowledge transfered from Socrates to *Socrates.

The program was developed on a very tight schedule. Dailey implemented a bare-bones chess program that copies chess boards and provided it to the MIT contingent in May 1994. During June, Dailey visited MIT to help tune the program, but we spent most of June simply getting the parallel version of the program to work correctly. The program started playing predictably only a few days before the tournament. The tournament was to start on Saturday morning, and on the previous Thursday night the program crashed 2 out 3 times that we played it. Friday morning we packed up two X-terminals and two modems into the trunk of our cars and drove the eight hours to Cape May, New Jersey, wondering whether we were going to be embarrassed by a program that would crash during tournament play. Friday night we logged in and made changes to the program until 3am. Then the tournament began. Saturday morning we played and won our first game. We noticed some problems with the program, and modified it for the Saturday evening match, which we also won. Saturday night we made some more modifications to the program, and on Sunday morning we won our third game. We left the program alone for the Sunday evening game, which we lost to Deep Thought. *Socrates's insufficient appreciation of the value of castling rights resulted in a poor move that Deep Thought punished brilliantly in what the on-site commentators called "one of the all-time greatest games of computer chess". Our fifth game resulted in a disappointing loss to Zarkov, in which *Socrates made two mistakes due to insufficient chess knowledge. The first mistake was similar to the mistake in the game against Deep Thought, but *Socrates managed to salvage the game to a drawn rook and pawn endgame. Unfortunately, *Socrates managed to find a losing move in a position that the commentators thought was nearly a forced draw. Throughout the tournament the program ran without crashing, and searched quite deeply. If only we had given Dailey more time to tune the chess knowledge...

One of the important organization differences between *Tech and *Socrates is that *Socrates separates the application from the scheduler, whereas in *Tech the scheduler and the application were wound up together. More importantly, *Socrates employs a linguistic layer to help the programmer express the program independently of the scheduler. Separating the system greatly simplified the implementation of *Socrates, and allowed us to implement several other parallel applications including a protein folding program [PJG*94] which was the first program to find the number of Hamiltonian paths in a $4 \times 4 \times 3$ grid, and some smaller programs such as the doubly recursive Fibonacci routine, a backtracking search to solve the problem of determining how many ways there are to place $n$ queens on an $n$ by $n$ chess board, a ray-tracing image rendering program, and a radiosity image rendering program.

We are now developing additional mechanisms for Cilk to provide high performance on a wider variety of applications. We are trying to improve the linguistic layer, to develop abstractions for manipulating shared data structures, and to simplify the interface to input/output and the operating system.

## Acknowledgments

Charles E. Leiserson, Robert D. Blumofe, Yuli Zhou, and Michael Halbherr all contributed to making the chess program work and to developing the underlying parallel technology used in *Socrates. Don Dailey and Larry Kaufman of Heuristic Software provided the serial program, Socrates, on which our parallel program is based, and Don worked many hours to help us get our parallel program working. Hans Berliner and Chris McConnell of CMU provided the serial version of Hitech that we first used as a testbed to develop our ideas for parallel game tree search. The National Center for Supercomputing Applications at the University of Illinois at Urbana-Champagne (NCSA) provided a 512-processor CM-5 for both the 1993 and the 1994 ACM Computer Chess Championships under NCSA Grant TRA930289N.

## 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.

[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.)

[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.

[BJK*94] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Phil Lisiecki, Keith H. Randall, Andy Shaw, and Yuli Zhou. *Cilk 1.1 Reference Manual.* Massachusetts Institute of Technology, Laboratory for Computer Science, September 1994. (Available via anonymous FTP from theory.lcs.mit.edu in /pub/cilk/manual1.0.ps.Z.)

[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.)

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM,* **21** (2), pages 201–206, April 1974.

[BB94] 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 via anonymous FTP from ftp.lcs.mit.edu in /pub/supertech/strata.)

[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.)

[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.)

[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.

[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, April 1994, 22 pp. (A shorter version will appear in Proc. of 2nd Int. Workshop on Massive Parallelism: Hardware, Software and Applications. Capri, Italy, Oct. 1994.)

[HL93] Robert V. Hogg and Johanenes Ledolter. *Applied Statistics for Engineers and Physical Scientists.* Macmillan Publishing Company, New York, 1993.

[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.

[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.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence,* **6** (4), pages 293–326, Winter 1975.

[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 csg-ftp.lcs.mit.edu in /pub/users/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.

[McA88] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence,* **35**, pages 287–310, 1988.

[PJG*94] Vijay S. Pande, Chris Joerg, Alexander Yu Grosberg, and Toyoichi Tanaka. Enumeration of the Hamiltonian walks on a cubic sublattice. *Journal of Physics A.,* 1994. (To appear.)

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

[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.