Using Cilk to Write Multiprocessor Chess Programs

Don Dailey
Charles E. Leiserson
MIT Laboratory for Computer Science

September 27, 2001

Cilk (pronounced "silk") is a C-based, algorithmic, multithreaded language for parallel programming developed at the MIT Laboratory for Computer Science. Cilk makes it easy to program irregular parallel applications such as chess. The Cilk programmer need not worry about protocols, job coordination, and load balancing, since they are handled automatically by Cilk's runtime system using provably efficient mechanisms. Cilk has been used to program a variety of multiprocessor chess programs, including the award-winning ⋆Socrates and Cilkchess programs. This paper overviews the Cilk language, illustrating how Cilk supports the programming of parallel game-tree search and other chess mechanisms.

1 Introduction

The Supercomputing Technologies (Supertech) Research Group in the MIT Laboratory for Computer Science began developing the Cilk multithreaded language [5, 8, 22, 27] in 1994. Development of Cilk has been intertwined with the development of a series of computer chess programs: StarTech, ⋆Socrates, and Cilkchess. Although the development of Cilk itself has been funded by the U.S. Defense Advanced Research Projects Agency (DARPA), all of our chess programs have been "skunkworks," developed in our spare time without research support.[1] Nevertheless, over the years, computer chess has provided much of the impetus for evolving the Cilk language.

Our first parallel chess program was StarTech [30, 31], written by Bradley Kuszmaul. StarTech's evaluation function is a software version of Hans Berliner's serial Hitech program [4]. The parallel search algorithm, which Kuszmaul called "Jamboree search," uses the "young brothers wait" heuristic [18] to parallelize Scout search [34]. StarTech won Third Prize in the 1993 ACM International Computer-Chess Championship [29] running on a 512-node Connection Machine CM5 at the University of Illinois National Center for Supercomputer Applications (NCSA).

[1]The MIT Laboratory for Computer Science has made discretionary funds available for us to enter our chess programs in computer-chess tournaments around the world. We are grateful for this support.

Our experience with StarTech convinced us that the logic of the game-tree search itself should be separated from the logic of scheduling, load balancing, and job coordination. In StarTech, the application and system logic are intermixed in an event-driven state machine, producing obscure code that was undebuggable by anyone but someone of Kuszmaul's caliber. Fortunately, the Supertech group's research focus was soon to provide better technology.

At the time Kuszmaul was working on StarTech, two collateral research efforts were underway in the MIT Laboratory for Computer Science. Michael Halbherr, Chris Joerg, and Yuli Zhou were developing a multithreaded language called PCM [25] within the dataflow research group led by Professor Arvind. In addition, Robert Blumofe and Charles Leiserson of the Supertech group were investigating algorithms for scheduling multithreaded computations [9, 10]. The two teams combined forces to develop the first version of a runtime system we called Cilk. Cilk-1 uses the same C preprocessing front-end as PCM, but it incorporates a provably good "work-stealing" scheduler based on the ideas of Blumofe and Leiserson.

While designing Cilk-1, the Supertech team started to work with Don Dailey and Larry Kaufman on a new chess program: ⋆Socrates [26]. This program was based on Dailey and Kaufman's serial Socrates program, which had won First Prize in the 1993 ACM International Computer-Chess Championship [29], where StarTech won Third Prize. Parallelizing the serial algorithm with Cilk-1 took about a month and a half of effort, most of which was done by Dailey and Joerg. Eventually, ⋆Socrates won Third Prize in the 1994 ACM International Computer-Chess Championship [33] running on NCSA's 512-node CM5. Later, ⋆Socrates won Second Prize in the 1995 ICCA World Computer-Chess Championship [40] in Hong Kong running on the 1824-node Intel Paragon at Sandia National Laboratories in New Mexico, losing to the program Fritz in the tie-breaking playoff game.

Although Cilk-1 insulates the programmer from scheduling and other runtime issues, it is still a painful language to use, because the parallel-programming model, which is based on explicit continuation passing, demands that the programmer write difficult-to-debug protocols. Cilk-1 lacks subroutine-like call/return semantics, requiring instead that threads communicate control explicitly. Although one can "wire up" any parallel control structure in Cilk-1, the resulting code is so protocol-laden that for ⋆Socrates, only Joerg was capable of modifying it without introducing bugs. Ironically, we had overcome the limitations of StarTech, which enabled us to write far more complicated code, but which once again we could not debug. Although we had made strides in allowing applications to be coded without worrying about scheduling, Cilk-1 still requires the programmer to engineer a rat's nest of protocols for threads to communicate. We needed a way of writing protocol-free, or at least largely protocol-free, multithreaded code.

Our second version of Cilk provides a call/return semantics for parallelism using simple `spawn` and `sync` keywords, features that remain in today's Cilk-5. Instead of being a simple C preprocessor, Rob Miller implemented Cilk-2's compiler `cilk2c` [32] as a type-checking source-to-source translator which compiles a Cilk source into a C postsource. The C postsource is then run through an ordinary C compiler and linked with the Cilk runtime system to produce object code. Cilk-2 was a resounding success. Its call/return parallelism simplified the coding of many applications, including graphics rendering by ray tracing and protein folding by backtrack search.

The one application that we found ourselves unable to code in "pure" Cilk-2, however, was computer chess. One reason was that computer chess requires a global transposition table in which previously evaluated positions are stored. Since ⋆Socrates was developed for distributed-memory parallel computers, such as the Connection Machine CM5 and Intel Paragon, it keeps its transposition table stored across the individual memories of processors. These platforms lack hardware support for shared memory. Since Cilk-2 provides no software remedy, ⋆Socrates must store and look up chess positions explicitly using the Strata [12] "active" message-passing layer. Although the active messaging is encapsulated within the software module for the transpostion table, we were disappointed that chess could not be programmed without resorting to a protocol layer.

Cilk-3 tried to address this shared-memory issue by providing software distributed shared memory, supported by the `cilk2c` compiler. Because Cilk-3 operates on large virtual-memory pages, however, it supports a relaxed model of consistency [7, 6]. Although Cilk-3's consistency model allowed us to write applications such as matrix multiplication and LU-decomposition, the model does not support the kind of fine-grained shared access of large tables needed by a chess application.

In addition, although call/return semantics allow the control for a wide variety of programs to be easily coded, the nondeterministic parallel searching algorithms required by computer chess cannnot be coded. The Supertech group debated whether the research progress on Cilk-2 and Cilk-3 had been wasted for the computer-chess application, since neither call/return semantics nor software distributed shared memory had provided any real answer to the complexities of coding computer chess.

Part of the answer was provided by hardware vendors. Thanks to a generous donation in 1996 by Sun Microsystems, the Supertech research group obtained access to a 12-processor Enterprise 5000 system, a "symmetric multitprocessor" providing consistent shared memory. With this computer system, coding a chess transposition table was trivial, since each computer could directly access all of memory without interrupting another processor.

Research provided the rest of the answer. Inspired by work at Berkeley [15], Cilk-4 introduced the "inlet" concept into Cilk-4. An inlet is a linguistic mechanism which facilitates coding of the nondeterministic search required for parallel game-tree searching (see Section 2). In addition to inlets, Cilk-4 provides an "abort" feature to allow speculative computations to be terminated when it is determined that they are no longer needed. With these features, programming parallel nondeterministic applications, such as chess, becomes much easier.

Keith Randall led the implementation of Cilk-4, which was the first version of Cilk designed for shared-memory multiprocessors. Eventually, Matteo Frigo engineered a major rewrite of Cilk-4 to produce the more stable and maintainable Cilk-5 release. Volker Strumpen also contributed to enhancing the robustness of the system, and many others wrote software components. The current Cilk-5 release runs on most shared-memory multiprocessors (Silicon Graphics, Sun, Digital/Compaq, Intel, etc.).

Our latest chess program Cilkchess was written with help from many of the members of the Supertech group. Cilkchess uses the inlet feature of Cilk-5 to implement a parallel version of the MTD(f) search algorithm [36]. Cilkchess won First Prize at the 1996 Dutch Open Computer-Chess Championship [41] running on MIT's 12-processor Sun Enterprise 5000, and Second Prize in 1997 [42] and 1998 [43] running on Boston University's 64-processor Silicon Graphics Origin 2000. At the 1999 World Computer-Chess Championship [17], running on a 256-processor SGI Origin 2000 at NASA Ames, Cilkchess ended up fourth out of a field of 30 programs, 1/2 point behind the program Shredder, which became World Champion as a result of the tournament.

The remainder of this paper illustrates how Cilk supports the programming of multiprocessor chess programs. Section 2 overviews Cilk's linguistic mechanisms. Section 3 describes how the performance of Cilk programs can be modeled, using the ⋆Socrates chess program for illustration. Section 4 shows how Cilk supports the programming of a chess search algorithm. Section 5 explores how Cilk supports other aspects of chess programming, including transposition tables and repetition testing. Finally, Section 6 offers some concluding remarks.

2 The Cilk language

The Cilk multithreaded language consists of C augmented by five new keywords to indicate parallel control. This section overviews the Cilk language, explaining the role of each of keyword in the programming of parallel applications.

Figure 1 shows a Cilk program that computes the nth Fibonacci number.[2] The program uses three Cilk keywords: `cilk`, `spawn`, and `sync`. Observe that if these keywords are deleted, a syntactically and semantically correct C program results, which we call the ***C elision*** of the Cilk program. Cilk is a ***faithful*** extension of C in that a Cilk program's C elision provides a legal implementation of the parallel semantics. Cilk introduces no new data types.

The keyword `cilk` identifies a Cilk procedure definition. A Cilk procedure is the parallel analog of a C function, having an argument list and body just like a C function. A Cilk procedure may spawn subprocedures in parallel and synchronize upon their completion.

Most of the work in a Cilk procedure is executed serially, just like C, but parallelism is created when the invocation of a Cilk procedure is immediately preceded by the keyword `spawn`. A spawn is the parallel analog of a C function call, and like a C function call, when a Cilk procedure is spawned, execution proceeds to the child. In an ordinary C function call, the parent is not resumed until after its child returns. In contrast, a Cilk spawn allows the parent to continue to execute in parallel with the child. Indeed, the parent can continue to spawn off children, producing a high degree of parallelism. Cilk's scheduler takes the responsibility of scheduling the spawned procedures on the processors of the parallel computer.

A Cilk procedure cannot safely use the return values of the children it has spawned until

[2]This program uses an inefficient, exponential-time algorithm. Although logarithmic-time methods are known [14, page 850], this program nevertheless provides a good didactic example.

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

cilk int fib (int n)
{
  if (n<2)
  {
    return(n);
  }
  else
  {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}

cilk int main (int argc, char *argv[])
{
  int n, result;
  n = atoi(argv[1]);
  result = spawn fib(n);
  sync;
  printf ("Result: %d\n", result);
  return 0;
}
```

Figure 1: A parallel Cilk program to compute the nth Fibonacci number.

it executes a `sync` statement. If all of its children have not completed when it executes a `sync`, the procedure suspends and does not resume until all of its children have completed. The `sync` statement is a local "barrier," not a global one as, for example, is sometimes used in message-passing programming. In Cilk, a `sync` waits only for the spawned children of the procedure to complete, not for the whole world. When all of its children return, the procedure resumes execution at the point immediately following the `sync` statement. In the Fibonacci example, a `sync` statement is required before the statement `return (x+y)` to avoid the anomaly that would occur if `x` and `y` were summed before both had been computed. A Cilk programmer uses the `spawn` and `sync` keywords to expose the parallelism in a program, and the Cilk runtime system takes the responsibility of scheduling the execution of the procedures efficiently.

Cilk's runtime system supports C's semantics for stack-allocated storage. A pointer to a local variable can be passed to a subroutine, but a pointer to a local variable cannot be returned, since local variables are deallocated automatically on a return. Cilk supports exactly these semantics, while allowing subprocedures to execute in parallel. In addition, Cilk supports heap memory through a `malloc()` function.

Most parallel applications can be programmed in Cilk using only the `cilk`, `spawn`, and `sync` keywords, but some tasks, such as chess, require speculative work to be done. For example, a search may spawn off two subsearches in parallel, only to discover that one of these searches returns a clear result, thereby making the other search irrelevant. Cilk provides two additional keywords — `inlet` and `abort` — which allow such nondeterministic programs to be coded. These five keywords make up the entirety of the Cilk language. Other parallel-programming mechanisms, such as locks for mutual exclusion, are provided through library functions. We shall first explain inlets, and then the abort mechanism.

Cilk's ***inlet*** feature provides flexibility in how values are returned from a child to its parent. Ordinarily, the valued returned by a spawned procedure is stored into a variable in its parent's frame:

```
x = spawn foo(y);
```

An inlet allows the returned value to be incorporated into its parent's frame in a more complex way.

An inlet is essentially a C function internal to a Cilk procedure.[3] Normally in Cilk, the spawning of a procedure must occur as a separate statement and not in an expression. An exception to this rule is made if the spawn is performed as an argument to an inlet. In this case, the procedure is spawned, and when it returns, the inlet is invoked. In the meantime, control of the parent procedure proceeds to the statement following the inlet.

Figure 2 illustrates how the `fib()` function can be coded using an inlet. The inlet `summer()` is defined to take a returned value `result` and add it to the variable `x` in the frame of the procedure that does the spawning. All the variables of `fib()` are available within `summer()`, since it is an internal function of `fib()`.

[3] If a Cilk program contains inlets, its C elision contains internal functions, which are not allowed in ANSI C. Cilk is based on Gnu C, however, which does permit internal functions.

```
cilk int fib (int n)
{
  int x = 0;

  inlet void summer (int result)
  {
    x += result;
    return;
  }

  if (n<2)
  {
    return n;
  }
  else
  {
    summer(spawn fib (n-1));
    summer(spawn fib (n-2));
    sync;
     return (x);
  }
}
```

Figure 2: Computing the nth Fibonacci number using an inlet.

Ensuring proper semantics for a program can be difficult if several inlets of a procedure, and possibly the procedure itself, update the same variables simultaneously. To ease the programming of these interactions, Cilk guarantees that these logically parallel "threads" operate atomically with respect to one another. In other words, when updating variables in the procedure frame, an inlet need not worry that frame variables are being simultaneously updated by the procedure itself or by another inlet. This implicit atomicity makes it fairly easy to reason about concurrency involving the inlets of a procedure instance without locking, declaring critical regions, or the like.

Cilk's `abort` keyword allows speculative work to be aborted without waiting for it to complete. The `abort` statement, when executed inside an inlet, causes all of the already-spawned descendants of the procedure to terminate immediately. Cilk takes the responsibility of hunting down all the descendants and terminating them.

As an example, suppose that a search spawns off two subsearches in parallel, and each subsearch returns its results via an inlet. If the result of one of the subsearches obviates the need to continue executing the other subsearch, Cilk's abort mechanism can be used to terminate it. We shall see in Section 4 how the `abort` statement eases the programming of parallel alpha-beta search.

The Cilk-5 reference manual [24] provides complete documentation of the Cilk language.

3 Cilk performance

Cilk supports an algorithmic programming model for parallel computation. Specifically, Cilk guarantees that programs are scheduled efficiently by its runtime system. This guarantee enables algorithms to be designed whose performance can be predicted analytically. In this section, we overview Cilk's performance model. We illustrate how $\star$Socrates allowed us to validate this model and how in turn, Cilk's performance model allowed us to make intelligent decisions about the design of the chess program.

Modeling Cilk program execution

A Cilk program execution consists of a collection of procedures—technically, procedure instances—each of which is broken into a sequence of nonblocking "threads." In Cilk terminology, a ***thread*** is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` statement. The first thread that executes when a procedure is called is the procedure's initial thread, and the subsequent threads are successor threads. At runtime, the binary "spawn" relation causes procedure instances to be structured as a rooted tree, and the dependencies among their threads form a directed acyclic graph (dag) embedded in this spawn tree, as is illustrated in Figure 3.

A correct execution of a Cilk program must obey all the dependencies in the dag, since a thread cannot be executed until all the threads on which it depends have completed. These dependencies form a partial order, permitting many ways of scheduling the threads in the dag. The order in which the dag unfolds and the mapping of threads onto processors are

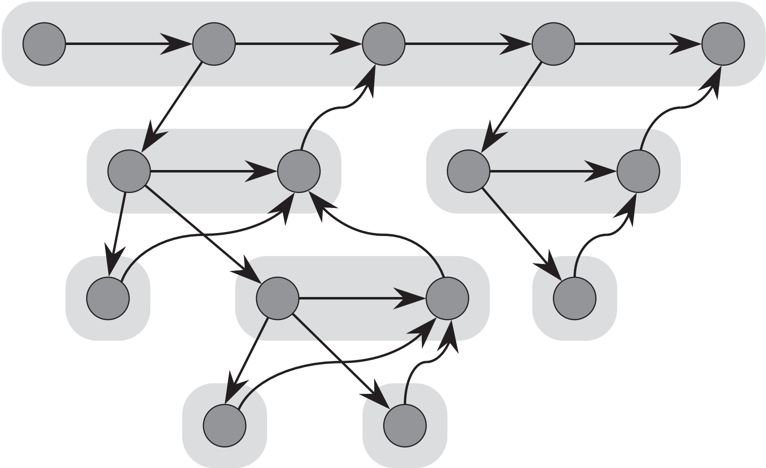

Figure 3: The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. All three types of edges are dependencies that constrain the order in which threads are scheduled.

crucial decisions made by Cilk's scheduler. Every active procedure has associated state that requires storage, and every dependency between threads assigned to different processors requires communication. Thus, different scheduling policies can yield different space and time requirements for the computation.

It can be shown that for general multithreaded dags, no good scheduling policy exists. That is, a dag can be constructed for which any schedule that provides linear speedup also requires vastly more than linear expansion of space [9]. Fortunately, every Cilk program generates a well-structured dag which can be scheduled efficiently [10].

The Cilk runtime system implements a provably efficient scheduling policy based on randomized work-stealing. During the execution of a Cilk program, when a processor runs out of work, it asks another processor chosen at random for work to do. Locally, a processor executes procedures in ordinary serial order (just like the C language's runtime system does), exploring the spawn tree in a depth-first manner. When a child procedure is spawned, the processor saves local variables of the parent on the bottom of a stack and commences work on the child. When the child returns, the bottom of the stack is popped (just like C) and the parent resumes. When another processor requests work, however, work is stolen from the top of the stack, that is, from the end opposite that which is normally used.

Performance modeling

Cilk's work-stealing scheduler executes any Cilk computation in nearly optimal time. From an abstract theoretical perspective (discounting cache effects and other phenomena that lead to superlinear speedups), there are two fundamental limits as to how fast a Cilk program can run. Let us denote by T_P the execution time of a given computation on P processors. The ***work*** of the computation is the total time needed to execute all threads in the dag. We can denote the work by T_1, since the work is essentially the execution time of the computation on one processor.

The first limit dictates that with T_1 work and P processors, the lower bound

$$T_P \geq T_1/P \tag{1}$$

must hold. The reason is that in one step, at most P work can be done by the P processors. Consequently, to do all of the T_1 work, it must take at least T_1/P time. The second limit is based on the program's ***critical-path length***, denoted by T_∞, which is the execution time of the computation on an infinite number of processors, or equivalently, the time needed to execute threads along the longest path of dependency.

The second lower bound is simply

$$T_P \geq T_\infty \ . \tag{2}$$

This bound says that a finite number of processors cannot execute the computation faster than an infinite number of processors.

Cilk's randomized work-stealing scheduler executes a Cilk computation on P processors in expected time

$$T_P \leq T_1/P + O(T_\infty) \ ,$$

which is asymptotically optimal. Empirically, the constant factor hidden by the big O is often close to 1 or 2 [8], and the formula

$$T_P \approx T_1/P + T_\infty \ , \tag{3}$$

which resembles "Brent's theorem" [11], is a good approximation of runtime. Cilk provides automatic timing instrumentation that can calculate the measures of work and critical-path length during program execution, thus allowing programmers to predict performance across the range of possible machine sizes. Moreover, Cilk has been engineered so that the cost of spawning is only 2–6 times the cost of an ordinary C function call, the actual value depending on the particular computer platform. Since the number of spawns performed by a real program during runtime tends to be relatively small, spawns have a negligible impact on running time. The low cost of spawns encourages Cilk programmers to think about spawning as a natural and inexpensive way to expose parallelism in their applications.

The performance model provided by Equation (3) can be interpreted using the notion of ***parallelism***, which is defined as $\overline{P} = T_1/T_\infty$. The parallelism is the average amount of work for every step along the critical path. Whenever $P \ll \overline{P}$, meaning that the actual number of processors is much smaller than the parallelism of the application, we have equivalently that $T_1/P \gg T_\infty$. Thus, the model predicts that $T_P \approx T_1/P$ and the Cilk program runs with almost perfect linear speedup.

Of course, the degree to which Equation (3) accurately predicts the performance of an application depends on how "ideal" the machine is on which the application is run.

If the machine has inadequate memory bandwidth, for example, performance will suffer compared what is predicted by this performance model. As it turns out, however, we have found Equation (3) to be an excellent predictor of performance over a wide range of parallel computers.

Using chess to benchmark Cilk's scheduler

In an early paper on Cilk [8], we used our $\star$Socrates chess program to document the efficacy of Cilk's scheduler on the Connection Machine CM5 parallel computer. Figure 4 shows a graph borrowed from that paper. The figure shows the outcome from many experiments of running $\star$Socrates on a variety of chess positions using various numbers of processors. Each "+" symbol in the figure indicates the measured speedup W_P/T_P for a P-processor run against the machine size P for that run, where W_P is the work of the computation.

For clarity in this discussion, we denote the work in a P-processor computation by W_P, rather than by T_1 as we have done thus far, because $\star$Socrates uses a "speculative" search algorithm. Recall that as we have defined the term "work," it is the total time needed to execute all the threads in the computation dag. For a deterministic parallel algorithm, the work of a program is the same, independent of the number of processors on which the program is run, and hence, it is accurate to use T_1 to represent the work. For a nondeterministic computation executed on several processors, however, the computation dag may vary from run to run. Consequently, the work W_P represented in a P-processor computation dag may bear little or no relation to the work T_1 of a serial execution. Since our goal is to evaluate the efficacy of Cilk's scheduler, we focus on the speedup W_P/T_P, because the normal speedup T_1/T_P incorporates work overhead produced by $\star$Socrates's speculative search algorithm for which Cilk's scheduler is not responsible.

In order to compare the outcomes for different runs, we have normalized each axis by dividing by the parallelism W_P/T_∞. Thus, a normalized machine size of 1.0 on the horizontal axis indicates a run where the parallelism equals the machine size. A normalized machine size of 0.1 indicates a run in which the parallelism exceeds the machine size by a factor of 10. On the vertical axis, a normalized speedup of 1.0 indicates a run that attains the maximum possible speedup. A normalized speedup of 0.1 indicates a run in which the speedup is 1/10 the maximum possible.

The two lower bounds (1) and (2) provide upper bounds on speedup, which can be interpreted as lines in Figure 4. The horizontal line at 1.0 is the upper bound on speedup obtained from the critical-path length, and the 45-degree line is the linear speedup bound. In addition, the curve for Equation (3) is plotted, and as can be seen from the figure, it interpolates the data reasonably well.

The figure shows that on runs for which the parallelism exceeds the number of processors, Cilk's scheduler obtains nearly perfect linear speedup. This region is where we normally would like an application to run, since otherwise the marginal return on an additional processor is diminishing. In the region where the number of processors is large compared to the parallelism, the data is more scattered, but the speedup is generally within a factor of

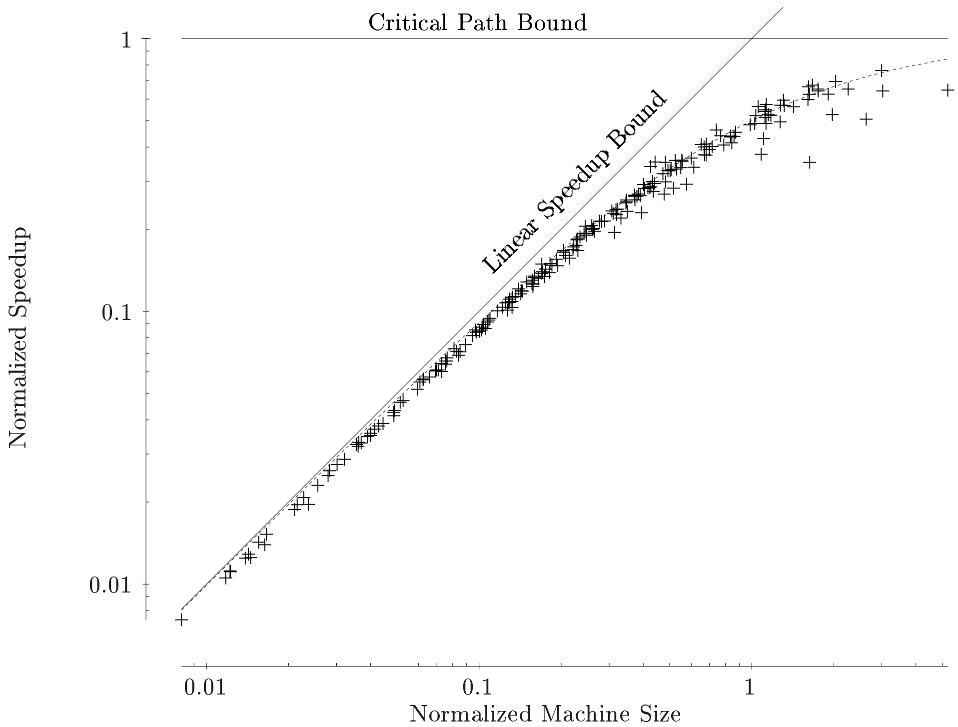

Figure 4: Normalized speedups for the ⋆Socrates chess program running on a Connection Machine CM5.

2 of the upper bound given by critical-path length. We prefer not to operate in this range, since the application could run nearly as fast with fewer processors. If the application does not exhibit good parallelism, however, it may be forced to operate in this domain. Fortunately, good chess algorithms exhibit a high degree of parallelism, and all of our chess programs operate in the linear-speedup regime most of the time. (Although we have spot-verified this performance model for later versions of Cilk, we have not repeated the extensive data-gathering.)

⋆Socrates speedup anomaly

The measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs, a feature we were able to exploit when designing the search algorithm for ⋆Socrates. For the the 1994 ACM International Computer Chess Championship, our program ran on NCSA's 512-node CM5. Because of the high demand for this massively parallel processor, one of the largest machines of its day, our access to it was limited. Consequently, we developed and tested most of our code on a 32-node CM5 at MIT.

During development, in an attempt to optimize ⋆Socrates performance, one of our programmers suggested a change to the search algorithm. We benchmarked the original version of the algorithm against the proposed version on the MIT machine, and we discovered the new version to be significantly faster. Nevertheless, we abandoned the proposed change, because our analysis of work and critical-path length indicated that it would be far slower on the much larger NCSA machine to be used in the tournament.

To understand this speedup anomaly, the numerical figures in the following scenario have been simplified for didactic purposes. The original program ran the benchmark in $T_{32} = 65$ seconds on the 32-node MIT machine. The proposed program ran the benchmark in $T'_{32} = 40$ seconds on the MIT machine. But these numbers tell us little about what might happen on the 512-node NCSA machine.

Fortunately, Cilk's instrumentation allowed us to analyze the situation. We discovered that the original program had work $T_1 = 2048$ seconds and critical-path length $T_\infty = 1$ second. The proposed program had work $T'_1 = 1024$ seconds and a much longer critical-path length of $T'_\infty = 8$ seconds. Using the model 3, we can verify that

$$\begin{aligned} T_{32} &= 2048/32 + 1 = 65 \ , \\ T'_{32} &= 1024/32 + 8 = 40 \ . \end{aligned}$$

In the actual incident, the agreement between model and experiment was close, but not exact. (We also coped with the nondeterminism of $\star$Socrates's search algorithm.)

With work and critical-path lengths in hand, we were now in a position to extrapolate the performance of the two algorithms on the 512-node machine:

$$\begin{aligned} T_{512} &= 2048/512 + 1 = 5 \ , \\ T'_{512} &= 1024/512 + 8 = 10 \ . \end{aligned}$$

The model predicts that on the tournament machine, the proposed change would slow performance by a factor of 2.

On the larger NCSA machine, the proposed program's longer critical-path length dominates the running time because of insufficient parallelism. The parallelism of the original program is $\overline{P} = T_1/T_\infty = 2048/1 = 2048$, whereas the parallelism of the proposed program is $\overline{P}' = T'_1/T'_\infty = 1024/8 = 128$. Thus, the proposed program runs out of parallelism on the 512-processor NCSA machine, whereas the original program still has some parallel "slackness" to exploit.

In the actual incident, the model predicted nearly a factor-of-3 degradation on the NCSA machine. Subsequent testing on the machine itself confirmed this prediction. Cilk's guarantee of efficient scheduling, together with the performance model it engenders, saved us from a costly design error.

4 Programming alpha-beta search

This section shows how a parallel version of alpha-beta search [28] can be programmed in Cilk. Although $\star$Socrates and Cilkchess use different parallel variants of minimax search based on Scout search [34] and MTD(f) [35], respectively, all the ideas in these parallel search algorithms can be illustrated in the context of alpha-beta search. This section outlines the basic "young-brothers-wait" strategy [18] for parallelizing alpha-beta search, and then it provides a walk-through of Cilk code that implements the strategy.

Since alpha-beta search is described in virtually every introduction to adversarial search (see, for example, [20, page 66] and [44, Chapter 6]), we shall assume basic familiarity with this search strategy. The key idea is that if White can make a move in a position which is so good that Black will not make the move leading to that position, then there is no point in searching White's other moves from that position. Those additional moves can be pruned. In order to get maximal pruning, therefore, it is advantageous to search the moves at a node in the search tree in best-first order.

The basic alpha-beta search algorithm is inherently serial, since it uses information from the search of one child of a node to prune subsequent children. When children are searched in parallel, however, it is hard to use information gained from searching one child to prune another. If one looks at an optimal game tree, however, one finds an interesting property: all of the nodes are either maximal (all of the children are searched) or singular (only one of the children is searched).

This observation suggests a parallel search strategy called ***young brothers wait*** [18]: if the first child of a node fails to generate a cutoff (the node is not singular), speculate that the node is maximal, and thus searching the remaining children in parallel wastes no work. To implement this strategy, the parallel alpha-beta algorithm first searches what it considers to be its best child, just like serial alpha-beta search. When that child returns, it may be that the alpha-beta algorithm prunes the rest of the children (a so-called beta-cutoff), and the search returns immediately. Otherwise, the algorithm speculates that the node is maximal, and it spawns off all the remaining children in parallel. If one returns with a score that causes a beta-cutoff, the other children are aborted, since their work has been rendered unnecessary.

We now walk through a Cilk implementation of this parallel search algorithm. The walk-through is broken into four parts. As will be seen, the code is minimally different from a C implementation of alpha-beta search, incorporating only six instances of Cilk keywords. The algorithm presented is a simple "full-width" search, shorn of mate and draw detection for didactic convenience. Mate and draw detection, as well as search heuristics such as null-move [2, 3, 16, 23] or killers [1, 39], can be incorporated into this code without difficulty, as has been done in both ⋆Socrates and Cilkchess.

The first part of the code defines the Cilk procedure `search`:

```
cilk int search( position *prev, int move, int depth )
{
  position cur;               /* current position       */
  int bestscore = -INF;       /* best score so far      */
  int num_moves;              /* number of children     */
  int mv;                     /* index of child         */
  int sc;                     /* child's score          */
  int cutoff = FALSE;         /* have we seen a cutoff? */
```

This code assumes that the current position is generated by the child, not the parent. Thus, a pointer to the parent position is passed in the parameter `prev`, and the current position will be produced by applying `move` to the parent position. The parameter `depth` is decremented by each recursive call until it becomes 0, so that the algorithm implements a full-width search. The `position` data structure contains fields `alpha` and `beta` delimiting the window of the search.

The second part of the code defines the inlet `catch`, which incorporates a child's score into the current node:

```
inlet void catch( int child_sc )
{
   child_sc = -child_sc;      /* negamax */
   if ( child_sc > bestscore )
   {
      bestscore = child_sc;
      if ( child_sc > cur.alpha )
      {
         cur.alpha = child_sc;
         if ( child_sc >= cur.beta )
         {
            cutoff = TRUE;     /* no need to search further */
            abort;             /* terminate other children  */
         }
      }
   }
}
```

The code implements a "negamax" [28] strategy wherein scores are always viewed from the point of view of the side to move. If the value `child_sc` returned by a child is the best so far, the variable `bestscore` is updated to record that fact. If the child's score exceeds the current value for `alpha`, then `cur.alpha` is updated. Finally, if the child's score equals or exceeds the current value for `beta`, a beta-cutoff occurs. The flag `cutoff` is set, which, as we shall see, will preclude future children from being spawned. In addition, children that have already been spawned are aborted.

The third part of the code is identical to an alpha-beta search in pure C, containing no Cilk keywords:

```
/* create current position and set up for search */
make_move( prev, &move, &cur );

sc = eval( &cur );   /* static evaluation */
if ( depth <= 0 )    /* leaf node */
{
```

```
    return( sc );
  }

  cur.alpha = -prev->beta;    /* negamax */
  cur.beta = -prev->alpha;

  /* generate moves, hopefully in best-first order */
  num_moves = gen_moves ( &cur );
```

The indicated move is made on the board, updating the `cur` structure with the new position. A static evaluation of the current position is made. If the current position is a leaf node of the search, because the desired depth of search has been achieved, the score from the static evaluation is returned. Otherwise, the alpha-beta window for the current search is updated, and the move generator is called.

The final part of the code performs the actual search:

```
  /* search the moves */
  for ( mv=0; !cutoff && mv<num_moves; mv++ )
  {
    catch( spawn search( &cur, mv, depth-1 ) );
    if ( mv==0 ) sync;   /* young brothers wait */
  }

  sync;   /* this sync is outside the loop so that the
             searches after the first execute in parallel */
  return( bestscore );
}
```

The loop spawns off the children of the current position. The loop guard terminates the loop if a child causes a beta-cutoff, which is discovered within the `catch` inlet. After the first child is spawned off, a `sync` is executed, suspending the loop until after the first child returns, thus implementing the young-brothers-wait strategy. The remaining children are spawned off in parallel, since no subsequent `sync` occurs within the loop. After all the children are spawned off, the algorithm syncs so that the best score of all the children can be returned.

Alpha-beta search makes a strong case for Cilk's efficient expressiveness. The difference between an ordinary C program for alpha-beta search and the Cilk program is only six keywords. Indeed, if minimizing the number of instances of Cilk keywords were the goal, the final `sync` could be eliminated by incorporating it into the `sync` within the loop. The code would be more cryptic, however.

The code for minimaz search in a high-performance chess program is far more complicated than the simple alpha-beta algorithm presented here. Among the major issues faced in a real program is how to minimize the likelihood that speculative code is executed futilely. Any branches of the search tree that are pruned by a serial search algorithm represent wasted work. When Cilkchess runs on a large multiprocessor, for example, the work can expand by a factor of 3. Good move-ordering heuristics tend to minimize the expansion of work.

5 Other parallel programming issues

In this section, we explore several other issues that arise when using Cilk to program a chess program. First, we examine how Cilk's locking primitives support atomic accesses to a transposition table. Second, we investigate how Cilk's shared-memory programming model simplifies the problem of discovering draws by repetition.

Cilk support for atomicity

All modern chess programs keep a ***transposition table*** to store positions that the program has seen. A chess position is entered into the transposition table when the search of that position returns a score. The entry for each position typically includes the depth that the position was searched, a bound on the score, move-ordering information, and various other heuristic and bookkeeping data. The idea is that if the program sees the same position in a later search, it may be able to use the information stored in the transposition table to improve the quality of the search or avoid the search altogether. Transposition tables are usually stored as large hash tables.

When two parallel threads access a common entry in a transposition table, anomalous behavior can result if one or both attempt to change the entry. This problem arises when the entry cannot be modified as a single atomic operation. While one thread is in the midst of changing several words of data, the other thread may see an entry consisting of both new (changed) and old (unchanged) data. To operate correctly, the second thread should see either the old data in its entirety or the new data in its entirety, but never a mixture.

Cilk provides mutual-exclusion locks to allow the creation of atomic regions of code. In Cilk, a lock has type `Cilk_lockvar`. The two operations on locks are `Cilk_lock` to test a lock and block if it is already acquired, and `Cilk_unlock` to release a lock. Both functions take an object of type `Cilk_lockvar` as a single argument. The lock object must be initialized using `Cilk_lock_init()` before it is used. The region of code between a `Cilk_lock` statement and the corresponding `Cilk_unlock` statement is called a ***critical section***.

The following code illustrates how Cilk's mutual exclusion locks can be used to enforce atomicity in a transposition table `ttab`.

```
typedef struct
{
  Cilk_lockvar lock;
  int          key;
  int          score;
  int          bestmove;
  int          depth;
```

```
  .
  .
  .
}
ttentry;

ttentry ttab[TTSIZE];

void init_ttab()
{
  int i;
  for (i=0; i<TTSIZE; i++)
  {
    Cilk_lock_init(ttab[i].lock);
  }
}

void update_entry( ttentry *e, int key, int score, ... )
{
  Cilk_lock(e->lock); /* begin critical section */

  e->key = key;
  e->score = score;
       .
       .
       .
  Cilk_unlock(e->lock); /* end critical section */
}
```

A Cilk lock is stored as part of each transposition-table entry. Cilk locks must be initialized before their first use. This initialization is performed by the C function `init_ttab()` which must be called before the transposition table is used. The function `update_entry()` updates the entry `e` atomically by acquiring `e->lock` before modifying the entry and releasing `e->lock` afterwards. Thus, if two threads simultaneously attempt to change the entry, they execute in sequence without interference.

Why use a lock for each entry rather than a single lock for the entire transposition table? After all, wouldn't a single lock be simpler and save space? A single-lock solution can indeed work effectively if the number of processors on which Cilk is run is small, but it does not scale well with the number of processors. The problem is that while a lock is held, every other thread that attempts to acquire the lock must wait. As the number of threads increases, the lock becomes a bottleneck, causing time to be wasted by threads waiting for the lock to be released. In contrast, using one lock per table entry yields a scalable solution. Since the number of entries is usually far larger than the number of active threads, the chances of two threads contending for a lock is small, and little time is wasted.

The Supertech research group argued long and hard about whether locks should be included in Cilk. Cilk-5 represents great progress over Cilk-1 in reducing the amount of protocol that a programmer must write. The protocol of acquiring and releasing locks, albeit simple, reverses that progress. Eventually, we decided that the practical need for atomicity outweighed the complexity of locks. Our decision was aided by the development of a debugging tool we call the Nondeterminator [13, 19].

The Nondeterminator finds ***data races*** in Cilk code. A data race occurs when two parallel threads, holding no locks in common, access the same memory location, and one of the threads modifies the location. Data races may be intended by the programmer, but they are more likely to be bugs. The Nondeterminator executes the Cilk code serially on a given input, using a novel data structure that keeps track of what threads operate logically in parallel. Every read and write by the program is instrumented to see if a date race exists. The Nondeterminator is not a verification tool, since it simulates actual execution on a given input, but it does provide a guarantee of finding races if they exist.

At this point, we must confess that Cilkchess does contain a data race. We have described how Cilk's library functions for locking can be used to make accesses to the transposition table atomic. Cilkchess, however, does not lock accesses to the transposition table. We decided that the overhead for locking would actually weaken the program more on average than if we did no locking. We indeed risk that a race might occur, but we have determined that the odds that it actually would affect the outcome of a competition is negligible. Thus, Cilkchess is provably, and intentionally, non-bug-free.

Parallel testing for repetitions

The rules of chess allow a player to claim a draw if his move brings about the third repetition of a position (with the same side to move). Computer programs usually implement this rule during a search by considering any position that matches an ancestor in the game tree to be a draw. A common implementation of this strategy is to use a hash table for bookkeeping. We now examine why this hash-table strategy breaks down in a parallel implementation and how Cilk's shared-memory semantics allow repetitions to be easily detected in parallel.

The hash-table approach to repetition testing is fairly simple. All positions that have actually been played in the game are entered into the hash table before the search begins. During the search, whenever a position is encountered, it is entered into the hash table. The children of the position are then recursively searched. Scores are backed up in accordance with the minimax search algorithm to produce a score for the position. When the score has been computed, the position is removed from the hash table. During the recursive search, if a position is encountered that is already in the hash table, a repeated position has occurred.

In a parallel chess program, a naive implementation of this strategy fails to work. First, the problems of atomically updating the hash table must be solved, but that is not the main difficulty. When a thread comes across a position stored in the hash table, the thread does not know if the position was encountered by its ancestor — a real repetition — or by another thread exploring another part of the game tree that just happened to examine the same position. One can imagine keeping track of which game-tree nodes are associated with which thread of execution, but such bookkeeping schemes quickly become unwieldy.

To detect repeated positions, Cilkchess uses a method that parallelizes easily. For every board position, a pointer is maintained to the parent's position. When evaluating a position, Cilkchess walks the chain of ancestors from the current position backwards to the beginning of the game and compare to see if the same position already appears in the chain. Cilkchess actually checks only every second position starting from the fourth back, since a repeated position must have the same side on move and two consecutive positions with the same side on move cannot repeat. (Cilkchess also accelerates the process by comparing the hash keys of the positions.)

It might seem that the cost of scanning ancestors could become large, but the scan can usually be terminated quickly. Some moves, like captures or pawn pushes, have the characteristic that once played, no sequence of subsequent moves can bring about a position that existed before the execution of that move. These ***irreversible*** moves provide a barrier above which the scan need not explore. Our empirical studies of middle-game positions indicate that the number of ancestors that need to be checked is less than 2 on average, although this number increases slightly in the endgame. This strategy of scanning ancestors back to an irreversible move was used in the CHESS 4.5 program [39, page 103].

Because of Cilk's strong support for shared-memory semantics, the parallel Cilk code for the ancestor scan is identical to the serial C code. No parallel constructs whatsoever are needed. Although the scans of several threads may intersect at common ancestors, no locking or coordination is required, because the ancestor data structures are only being read, not modified. Some parallel languages require special mechanisms to dereference pointers to shared memory, but Cilk does not. Consequently, compared with the C implementation, the Cilk implementation incurs no undue performance penalty. The code is the same.

6 Conclusion

To produce high-performance parallel applications, programmers often focus on communication costs and execution time, quantities that are dependent on specific machine configurations. Cilk's philosophy argues that a programmer should think instead about work and critical-path length, abstractions that can be used to characterize the performance of an algorithm independent of the machine configuration. Cilk provides a programming model in which work and critical-path length are measurable quantities, and it delivers guaranteed performance as a function of these quantities. Moreover, Cilk programs "scale down" to run on one processor with nearly the efficiency of analogous C programs. Consequently, any performance tuning of the C elision of the Cilk program automatically accrue to the Cilk program itself.

The Supertech research group in LCS found that computer chess was an ideal vehicle for developing parallel-programming technology. Research groups can easily wander off solving abstract problems having no practical significance. Chess is a formidable real-world problem which long predates computers, not an artificial problem tailor-made to highlight our research. Chess challenged our research group to address previously neglected issues involving the parallel programming of irregular and symbolic computations, rather than regular and numerical problems that typify much of the existing literature on parallel computing. Students worked on an engaging problem that allowed them to highlight their parallel-programming research results in an externally visible way, (i.e., they could explain it to their parents). Tournament events motivated the team and gave us "end-to-end" [38] unbiased feedback on our work. Finally, chess programming was just plain fun!

The combination of chess and Cilk allowed the Supertech team to explore a gamut of issues across computer science. Too often, students come out of school believing that intellectual activity is segregated into disparate, nonoverlapping areas. Chess and Cilk allowed us to integrate knowledge in algorithms, artificial intelligence, programming languages, concurrency, computer architecture, software engineering, and high performance. The cross-pollination of ideas refined Cilk into a simple, but powerful, tool for parallel programming.

The Cilk developers are currently working on enhancing the Cilk system environment, including support for parallel I/O and streams, job scheduling, and fault tolerance. Cilk software, documentation, publications, and up-to-date information are available via the Web at `http://supertech.lcs.mit.edu/cilk`. Detailed descriptions of the foundation and history of Cilk can be found in [5, 27, 37, 21].

Acknowledgments

Many people have contributed to the series of Cilk chess programs produced by the Supertech research group of the MIT Laboratory for Computer Science. Special thanks go to Chris Joerg, who has continually provided solid input to every program, even after receiving his Ph.D. Others who have contributed over the years to our chess programs include Reid Barton, Don Beal, Bobby Blumofe, Matteo Frigo, Geoffrey Gelman, Runako Godfrey, Michael Halbherr, Larry Kaufman, Bradley Kuszmaul, Phil Lisiecki, Aske Plaat, Ryan Porter, Harald Prokop, Keith Randall, and Yuli Zhou. Thanks to Chris Joerg and Ernst Heinz for thoughtful comments on drafts of this manuscript. Finally, thanks to Michael Dertouzos, Director of MIT's Laboratory for Computer Science, for providing financial assistance so that our chess programs could compete in computer-chess competitions.

References

[1] S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *1977 ACM Annual Conference*, pages 466–473, Seattle, Washington. ACM.

[2] Don F. Beal. Experiments with the null move. In Don F. Beal, editor, *Advances in Computer Chess*, volume 5. Elsevier Science, 1989.

[3] Don F. Beal. A generalised quiesence search algorithm. *Artificial Intelligence*, 43(1):85–98, April 1990.

[4] Hans Berliner and Carl Ebeling. Pattern knowledge and search: the SUPREM architecture. *Artificial Intelligence*, 38(2):161–198, March 1989.

[5] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[6] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[7] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[9] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing*, pages 362–371, San Diego, California, May 1993.

[10] 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*, pages 356–368, Santa Fe, New Mexico, November 1994.

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

[12] Eric A. Brewer and Robert Blumofe. Strata: A multi-layer communications library. MIT Laboratory for Computer Science. Available as `ftp://ftp.lcs.mit.edu/pub/supertech/strata/strata.tar.Z`.

[13] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[15] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 1991.

[16] Christian Donninger. Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, September 1993.

[17] Mathias Feist. The 9th World Computer-Chess Championship. *ICCA Journal*, 22(3):149–159, September 1999.

[18] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. *Advances in Computer Chess 7*, pages 203–219, 1993.

[19] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, Newport, Rhode Island, June 1997.

[20] Peter W. Frey. An introduction to computer chess. In Peter W. Frey, editor, *Chess Skill in Man and Machine*, chapter 3, pages 54–81. Springer-Verlag, second edition, 1983.

[21] Matteo Frigo. *Portable High-Performance Programs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[22] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, June 1998.

[23] G. Goetsch and M.S. Campbell. Experiments with the null-move heuristic. In T.A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 159–168. Springer, 1990.

[24] Supercomputing Technologies Group. *Cilk 5.2 Reference Manual*. MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, July 1998. available on the World Wide Web at URL "`http://supertech.lcs.mit.edu/cilk`".

[25] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

[26] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 1994. Available as `ftp://theory.lcs.mit.edu/pub/cilk/dimacs94.ps.Z`.

[27] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

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

[29] Danny Kopec and Mike Valvo. The 23rd ACM International Computer-Chess Championship. *ICCA Journal*, 16(1):38–46, March 1993.

[30] Bradley C. Kuszmaul. *Synchronized MIMD Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645 or `ftp://theory.lcs.mit.edu/pub/bradley/phd.ps.Z`.

[31] Bradley C. Kuszmaul. The STARTECH massively parallel chess program. *ICCA Journal*, 18(1):3–19, March 1995.

[32] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[33] M. Newborn. The 24th ACM International Computer-Chess Championship. *ICCA Journal*, 17(3):159–164, September 1994.

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

[35] Aske Plaat. *Research Re: search & Re-search.* PhD thesis, Erasmus University, Rotterdam, The Netherlands, June 1996.

[36] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87:255–293, November 1996.

[37] Keith H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[38] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[39] David J. Slate and Lawrence R. Atkin. CHESS 4.5—the Northwestern University chess program. In Peter W. Frey, editor, *Chess Skill in Man and Machine*, chapter 4, pages 82–118. Springer-Verlag, second edition, 1983.

[40] H. K. Tsang and Don F. Beal. The 8th World Computer-Chess Championship. *ICCA Journal*, 18(2):93–111, June 1995.

[41] Th. van der Storm. Report on the 16th Open Dutch Computer-Chess Championship. *ICCA Journal*, 19(4):272–275, December 1996.

[42] Th. van der Storm. Report on the 17th Open Dutch Computer-Chess Championship. *ICCA Journal*, 20(4):271–272, December 1997.

[43] Th. van der Storm. Report on the 18th Open Dutch Computer-Chess Championship. *ICCA Journal*, 21(4):252–254, December 1998.

[44] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 1992.