Cilk: An Efficient Multithreaded Runtime System

Robert D. Blumofe  Christopher F. Joerg  Bradley C. Kuszmaul
Charles E. Leiserson  Keith H. Randall  Yuli Zhou
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

Cilk (pronounced “silk”) is a C-based runtime system for multithreaded parallel programming. In this paper, we document the efficiency of the Cilk work-stealing scheduler, both empirically and analytically. We show that on real and synthetic applications, the “work” and “critical path” of a Cilk computation can be used to accurately model performance. Consequently, a Cilk programmer can focus on reducing the work and critical path of his computation, insulated from load balancing and other runtime scheduling issues. We also prove that for the class of “fully strict” (well-structured) programs, the Cilk scheduler achieves space, time, and communication bounds all within a constant factor of optimal.

The Cilk runtime system currently runs on the Connection Machine CM5 MPP, the Intel Paragon MPP, the Silicon Graphics Power Challenge SMP, and the MIT Phish network of workstations. Applications written in Cilk include protein folding, graphic rendering, backtracking search, and the Socrates chess program, which won third prize in the 1994 ACM International Computer Chess Championship.

1 Introduction

Multithreading has become an increasingly popular way to implement dynamic, highly asynchronous, concurrent programs [1, 8, 9, 10, 11, 12, 15, 19, 21, 22, 24, 25, 28, 33, 34, 36, 39, 40]. A multithreaded system provides the programmer with a means to create, synchronize, and schedule threads. Although the schedulers in many of these runtime systems seem to perform well in practice, none provide users with a guarantee of application performance. Cilk is a runtime system whose work-stealing scheduler is efficient in theory as well as in practice. Moreover, it gives the user an algorithmic model of application performance based on the measures of “work” and “critical path” which can be used to predict the runtime of a Cilk program accurately.

A Cilk multithreaded computation can be viewed as a directed acyclic graph (dag) that unfolds dynamically, as shown schematically in Figure 1. A Cilk program consists of a collection of Cilk procedures, each of which is broken into a sequence of threads, which form the vertices of the dag. Each thread is a nonblocking C function, which means that it can run to completion without waiting or suspending once it has been invoked. As one of the threads from a Cilk procedure runs, it can spawn a child thread which begins a new child procedure. In the figure, downward edges connect threads and their procedures with the children they have spawned. A spawn is like a subroutine call, except that the calling thread may execute concurrently with its child, possibly spawning additional children. Since threads cannot block in the Cilk model, a thread cannot spawn children and then wait for values to be returned. Rather, the thread must additionally spawn a successor thread to receive the children’s return values when they are produced. A thread and its successors are considered to be parts of the same Cilk procedure. In the figure, sequences of successor threads that form Cilk procedures are connected by horizontal edges. Return values, and other values sent from one thread to another, induce data dependencies among the threads, where a thread receiving a value cannot begin until another thread sends the value. Data dependencies are shown as upward, curved edges in the figure. Thus, a Cilk computation unfolds as a spawn tree composed of procedures and the spawn edges that connect them to their children, but the execution is constrained to follow the precedence relation determined by the dag of threads.

The execution time of any Cilk program on a parallel computer with P processors is constrained by two parameters of the computation: the work and the critical path. The work, denoted \( T_w \), is the time used by a one-processor execution of the program, which corresponds to the sum of the execution times of all the threads. The critical path length, denoted \( T_{\text{crit}} \), is the total amount of time required by an infinite-processor execution, which corresponds to the largest sum of thread execution times along any path. With \( P \) processors, the

Figure 1: The Cilk model of multithreaded computation. Threads are shown as circles, which are grouped into procedures. Each downward edge corresponds to a spawn of a child, each horizontal edge corresponds to a spawn of a successor, and each upward, curved edge corresponds to a data dependency. The numbers in the figure indicate the levels of procedures in the spawn tree.
execution time cannot be less than \( T_1 / P \) or less than \( T_{\text{min}} \). The Cilk scheduler uses “work stealing” \([3, 7, 13, 14, 15, 19, 27, 28, 29, 34, 40]\) to achieve execution time very near to the sum of these two measures. Off-line techniques for computing such efficient schedules have been known for a long time \([5, 16, 17]\), but this efficiency has been difficult to achieve on-line in a distributed environment while simultaneously using small amounts of space and communication.

We demonstrate the efficiency of the Cilk scheduler both empirically and analytically. Empirically, we have been able to document that Cilk works well for dynamic, asynchronous, tree-like, MIMD-style computations. To date, the applications we have programmed include protein folding, graphic rendering, backtrack search, and the Socrates chess program, which won third prize in the 1994 ACM International Computer Chess Championship. Many of these applications pose problems for more traditional parallel environments, such as message passing \([38]\) and data parallel \([2, 20]\), because of the unpredictability of the dynamic workloads on processors. Analytically, we prove that for “fully strict” (well-structured) programs, Cilk’s work-stealing scheduler achieves execution space, time, and communication bounds all within a constant factor of optimal. To date, all of the applications that we have coded are fully strict.

The Cilk language is an extension to C that provides an abstraction of threads in explicit continuation-passing style. A Cilk program is preprocessed to C and then linked with a runtime library to run on the Connection Machine CM5 MPP, the Intel Paragon MPP, the Silicon Graphics Power Challenge SMP, or the MIT Phish \([4]\) network of workstations. In this paper, we focus on the Connection Machine CM5 implementation of Cilk. The Cilk scheduler on the CM5 is written in about 30 pages of C, and it performs communication among processors using the Strata \([6]\) active-message library.

The remainder of this paper is organized as follows. Section 2 describes Cilk’s runtime data structures and the C language extensions that are used for programming. Section 3 describes the work-stealing scheduler. Section 4 documents the performance of several Cilk applications. Section 5 shows how the work and critical path of a Cilk application is analyzed, and Section 6 concludes with some concluding remarks and describes our plans for the future.

2 The Cilk Programming Environment and Implementation

In this section we describe a C language extension that we have developed to ease the task of coding Cilk programs. We also explain the basic runtime data structures that Cilk uses.

In the Cilk language, a thread \( T \) is defined in a manner similar to a C function definition:

\[
\text{thread } T \{ \text{arg-decls} \ldots \} \{ \text{stmts} \ldots \}
\]

The Cilk preprocessor translates \( T \) into a C function of one argument and void return type. The one argument is a pointer to a closure data structure, illustrated in Figure 2, which holds the arguments for \( T \). A closure consists of a pointer to the C function for \( T \), a slot for each of the specified arguments, and a join counter indicating the number of missing arguments that need to be supplied before \( T \) is ready to run. A closure is ready if it has obtained all of its arguments, and it is waiting if some arguments are missing. To run a ready closure, the Cilk scheduler invokes the thread as a procedure using the closure itself as its sole argument. Within the code for the thread, the arguments are copied out of the closure data structure into local variables. The closure is allocated from a simple runtime heap when it is created, and it is returned to the heap when the thread terminates.

The Cilk language supports a data type called a continuation, which is specified by the type modifier keyword \texttt{cont}. A continuation is essentially a global reference to an empty argument slot of a closure, implemented as a compound data structure containing a pointer to a closure and an offset that designates one of the closure’s argument slots. Continuations can be created and passed among threads, which enables threads to communicate and synchronize with each other. Continuations are typed with the C data type of the slot in the closure. At runtime, a thread can spawn a child thread by creating a closure for the child. Spawning is specified in the Cilk language as follows:

\[
\text{spawn } T \{ \text{args} \ldots \}
\]

This statement creates a child closure, fills in all available arguments, and initializes the join counter to the number of missing arguments. Available arguments are specified as in C. To specify a missing argument, the user specifies a continuation variable (type \texttt{cont}) preceded by a question mark. For example, if the second argument is \( ?k \), then Cilk sets the variable \( k \) to a continuation that refers to the second argument slot of the created closure. If the closure is ready, that is, it has no missing arguments, then \texttt{spawn} causes the closure to be immediately posted to the scheduler for execution. In typical applications, child closures are usually created with no missing arguments.

To create a successor thread, a thread executes the following statement:

\[
\text{spawn \_next } T \{ \text{args} \ldots \}
\]

This statement is semantically identical to \texttt{spawn}, but it informs the scheduler that the new closure should be treated as a successor, as opposed to a child. Successor closures are usually created with some missing arguments, which are filled in by values produced by the children.

A Cilk procedure does not ever return values in the normal way to a parent procedure. Instead, the programmer must code the parent procedure as two threads. The first thread spawns the child procedure, passing it a continuation pointing to the successor thread’s closure. The child sends its “return” value explicitly as an argument to the waiting successor. This strategy of communicating between threads is called explicit continuation passing. Cilk provides primitives of the following form to send values from one closure to another:

\[
\text{send} \text{argument} \{ k, \text{value} \}
\]

This statement sends the value \texttt{value} to the argument slot of a waiting closure specified by the continuation \( k \). The types of the continuation and the value must be compatible. The join counter of the waiting
thread fib (cont int k, int n)
  { if (n<=2)
    send_argument (k, n)
  else
    { cont int x, y;
      spawn_next sum (k, x, y);
      spawn fib (x, n-1);
      spawn fib (y, n-2);
    }
  }

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

Figure 3: A Cilk procedure, consisting of two threads, to compute the n-th Fibonacci number.

closure is decremented, and if it becomes zero, then the closure is ready and is posted to the scheduler.

Figure 3 shows the familiar recursive Fibonacci procedure written in Cilk. It consists of two threads, fib and its successor sum. Reflecting the explicit continuation passing style that Cilk supports, the first argument to each thread is the continuation specifying where the "return" value should be placed.

When the fib function is invoked, it first checks to see if the boundary case has been reached, in which case it uses send_argument to "return" the value of n to the slot specified by continuation k. Otherwise, it spawns the successor thread sum, as well as two children to compute the two subcases. Each of these two children is given a continuation specifying to which argument in the sum thread it should send its result. The sum thread simply adds the two arguments when they arrive and sends this result to the slot designated by k.

Although writing in explicit continuation passing style is somewhat onerous for the programmer, the decision to break procedures into separate nonblocking threads simplifies the Cilk runtime system. Each Cilk thread leaves the C runtime stack empty when it completes. Thus, Cilk can run on top of a vanilla C runtime system. A common alternative [19, 25, 32, 34] is to support a programming style in which a thread suspends whenever it discovers that required values have not yet been computed, resuming when the values become available. When a thread suspends, however, it may leave temporary values on the runtime stack which must be saved, or each thread must have its own runtime stack. Consequently, this alternative strategy requires changes to the runtime system that depend on the C calling stack layout and register usage conventions. Another advantage of Cilk’s strategy is that it allows multiple children to be spawned from a single nonblocking thread, which saves on context switching. In Cilk, r children can be spawned and executed with only r + 1 context switches, whereas the alternative of suspending whenever a thread is spawned causes 2r context switches. Since our primary interest is in understanding how to build efficient multithreaded runtime systems, but without redesigning the basic C runtime system, we chose the alternative of burdening the programmer with a requirement which is perhaps less elegant linguistically, but which yields a simple and portable runtime implementation.

Cilk supports a variety of features that give the programmer greater control over runtime performance. For example, when the last action of a thread is to spawn a ready thread, the programmer can use the keyword call instead of spawn that produces a "tail call" to run the new thread immediately without invoking the scheduler. Cilk also allows arrays and subarrays to be passed as arguments to closures. Other features include various abilities to override the scheduler’s decisions, including on which processor a thread should be placed and how to pack and unpack data when a closure is migrated from one processor to another.

3 The Cilk work-stealing scheduler

Cilk’s scheduler uses the technique of work-stealing [3, 7, 13, 14, 15, 19, 27, 28, 29, 34, 40] in which a processor (the thief) who runs out of work selects another processor (the victim) from whom to steal work, and then steals the shallowest ready thread in the victim’s spawn tree. Cilk’s strategy is for thieves to choose victims at random [3, 27, 37].

At runtime, each processor maintains a local ready queue to hold ready closures. Each closure has an associated level, which corresponds to the number of spawn’s (but not spawn’s) on the path from the root of the spawn tree. The ready queue is an array in which the Lth element contains a linked list of all ready closures having level L.

Cilk begins executing the user program by initializing all ready queues to be empty, placing the root thread into the level-0 list of Processor 0’s queue, and then starting a scheduling loop on each processor. Within a scheduling loop, a processor first checks to see whether its ready queue is empty. If it is, the processor commences “work stealing,” which will be described shortly. Otherwise, the processor performs the following steps:

1. Remove the thread at the head of the list of the deepest nonempty level in the ready queue.
2. Extract the thread from the closure, and invoke it.

As a thread executes, it may spawn or send arguments to other threads. When the thread terminates, control returns to the scheduling loop.

When a thread at level L spawns a child thread T, the scheduler executes the following operations:

1. Allocate and initialize a closure for T.
2. Copy the available arguments into the closure, initialize any continuations to point to missing arguments, and initialize the join counter to the number of missing arguments.
3. Label the closure with level L + 1.
4. If there are no missing arguments, post the closure to the ready queue by inserting it at the head of the level-(L + 1) list.

Execution of spawn’s is similar, except that the closure is labeled with level L and, if it is ready, posted to the level-L list. A processor that executes send_argument(k, value) performs the following steps:

1. Find the closure and argument slot referenced by the continuation k.
2. Place value in the argument slot, and decrement the join counter of the closure.
3. If the join counter goes to zero, post the closure to the ready queue at the appropriate level.

When the continuation k refers to a closure on a remote processor, network communication ensues. The processor that initiated the send_argument function sends a message to the remote processor to perform the operations. The only subtlety occurs in step 3. If the closure must be posted, it is posted to the ready queue of the initiating processor, rather than to that of the remote processor. This policy is necessary for the scheduler to be provably good, but as a practical matter, we have also had success with posting the closure to the remote processor’s queue, which can sometimes save a few percent in overhead.

If the scheduler attempts to remove a thread from an empty ready queue, the processor becomes a thief and commences work stealing as follows:

1. Select a victim processor uniformly at random.
2. If the victim’s ready queue is empty, go to step 1.
3. If the victim’s ready queue is nonempty, extract a thread from
   the tail of the list in the shallowest nonempty level of the ready
   queue, and invoke it.

Work stealing is implemented with a simple request-reply communica-

tion protocol between the thief and victim.

Why steal work from the shallowest level of the ready queue? The

reason is two-fold. First, we would like to steal large amounts of work,
and shallow closures are likely to execute for longer than deep ones.
Stealing large amounts of work tends to lower the communication
cost of the program, because fewer steals are necessary. Second, the

closures at the shallowest level of the ready queue are also the ones that
are shallowest in the dag, a key fact proven in Section 6. Consequently,
if processors are idle, the work they steal tends to make progress along
the critical path.

4 Performance of Cilk applications

This section presents several applications that we have used to bench-
mark the Cilk scheduler. We also present empirical evidence from
experiments run on a CM5 supercomputer to document the efficiency
of our work-stealing scheduler. The CM5 is a massively parallel
computer based on 32MHz SPARC processors with a fat-tree inter-
connection network [30].

The applications are described below:

• fib is the same as was presented in Section 2, except that the
  second recursive spawn is replaced by a “tail call” that avoids
  the scheduler. This program is a good measure of Cilk overhead,
  because the thread length is so small.

• queens is a backtrack search program that solves the problem
  of placing \( N \) queens on a \( N \times N \) chessboard so that no two
  queens attack each other. The Cilk program is based on serial
code by R. Sargent of the MIT Media Laboratory. Thread length
  was enhanced by serializing the bottom ? levels of the search
tree.

• pfold is a protein-folding program [35] written in conjunc-
tion with V. Pande of MIT’s Center for Material Sciences and
Engineering. This program finds hamiltonian paths in a three-
dimensional grid using backtrack search. It was the first pro-
gram to enumerate all hamiltonian paths in a \( 3 \times 4 \times 4 \) grid. We
timed the enumeration of all paths starting with a certain
sequence.

• ray is a parallel program for graphics rendering based on the
  serial POV-Ray program, which uses a ray-tracing algorithm.
The entire POV-Ray system contains over 20,000 lines of C
  code, but the core of POV-Ray is a simple doubly nested loop
  that iterates over each pixel in a two-dimensional image. For
  ray we converted the nested loops into a 4-ary divide-and-
  conquer control structure using spawns.1 Our measurements
  do not include the approximately 2.4 seconds of startup time
  required to read and process the scene description file.

• knary \((k,n,r)\) is a synthetic benchmark whose parameters can be
  set to produce a variety of values for work and critical path.
  It generates a tree of branching factor \( k \) and depth \( n \) in which
  the first \( r \) children at every level are executed serially and
  the remainder are executed in parallel. At each node of the tree,
  the program runs an empty “for” loop for 400 iterations.

1 Initially, the serial POV-Ray program was about 5 percent slower than the Cilk version
  running on one processor. The reason was that the divide-and-conquer decomposition
  performed by the Cilk code provides better locality than the doubly nested loop of the
  serial code. Modifying the serial code to imitate the Cilk decomposition improved its
  performance. Timings for the improved version are given in the table.

2 For the ^Socrates program, \( T_{\text{ser}} \) is not the measured execution time, but rather it is
  an estimate of the work obtained by summing the execution times of all threads, which
  yields a slight underestimate. ^Socrates is an unusually complicated application, because
  its speculative execution yields unpredictable work and critical path. Consequently, the
  measured runtime on one processor does not accurately reflect the work on \( P > 1 \)
  processors.
for example, have in excess of 7000-fold parallelism and achieve more than 99 percent of perfect linear speedup on 32 processors and more than 95 percent of perfect linear speedup on 256 processors. The Socrates program exhibits somewhat less parallelism and also somewhat less speedup. On 32 processors the Socrates program executed over 17 million threads but migrated only 6170 (24.10 per processor) when run with 256 processors. Taking advantage of this property, other researchers [25, 32] have developed techniques for implementing spawns such that the aggregate average cost of a spawn/return in Cilk is between 8 and 9 times the cost of a function call/return in C.

Efficient execution of programs with short threads requires a low-overhead spawn operation. As can be observed from Table 4, the vast majority of threads execute on the same processor on which they are spawned. For example, the fib program executed over 17 million threads but migrated only 6170 (24.10 per processor) when run with 256 processors. Taking advantage of this property, other researchers [25, 32] have developed techniques for implementing spawns such that the child thread executes on the same processor as its parent, the cost of the spawn operation is roughly equal the cost of a C function call. We hope to incorporate such techniques into future implementations of Cilk.

Finally, we make two observations about the space and communication measures in Table 4. Looking at the “space/proc.” rows, we observe that the space per processor is generally quite small and does not grow with the number of processors. For example, Socrates on 32 processors executes over 26 million threads, yet no processor ever has more than 386 allocated space per processor. On 256 processors, the number of executed threads nearly doubles to over 51 million, but the space per processor barely changes. In Section 6 we show formally that for Cilk programs, the space per processor does not grow as we add processors.

Looking at the “requests/proc.” and “steals/proc.” rows in Table 4, we observe that the amount of communication grows with the critical path but does not grow with the work. For example, fib, queens,

\[
\begin{array}{cccccccc}
\text{application} & \text{fib} & \text{queens} & \text{pfold} & \text{ray} & \text{knary}^{(10, 5, 2)} & \text{knary}^{(10, 4, 1)} & \text{Socrates}^{(depth 10)} & \text{Socrates}^{(depth 10)} \\
\text{parameters} & (33) & (15) & (3, 3, 4) & (500, 500) & (10, 5, 2) & (10, 4, 1) & (32 proc.) & (256 proc.) \\
\hline
\text{T}_{\text{total}} & 8.487 & 252.1 & 615.1 & 729.2 & 288.6 & 40.993 & 1665 & 1665 \\
\text{T}_1 & 73.16 & 254.6 & 647.8 & 732.5 & 314.6 & 45.43 & 3644 & 7023 \\
\text{T}_{\text{total}}/T_1 & 0.116 & 0.9902 & 0.9496 & 0.9955 & 0.9174 & 0.9023 & 0.4569 & 0.2371 \\
\text{T}_1/T_{\text{total}} & 0.000326 & 0.0345 & 0.04354 & 0.0415 & 4.458 & 0.255 & 3.134 & 3.24 \\
\text{T}_1/T_{\text{proc}} & 224417 & 7380 & 14879 & 17650 & 70.56 & 178.2 & 1163 & 2168 \\
\text{threads} & 17,108,660 & 210,740 & 9,515,098 & 424,475 & 5,859,374 & 873,812 & 26,151,774 & 51,685,823 \\
\text{thread} & 4.276\mu s & 1208\mu s & 68.08\mu s & 1726\mu s & 53.69\mu s & 59.99\mu s & 139.3\mu s & 139.5\mu s \\
\text{efficiency} & 2.287 & 7.991 & 12.09 & 17.58 & 1.36 & 0.7933 & 0.9093 & - \\
\text{maximum} & 31.84 & 31.78 & 31.97 & 33.79 & 20.78 & 27.81 & 28.90 & 28.90 \\
\text{space/proc.} & 70 & 95 & 47 & 39 & 41 & 42 & 368 & - \\
\text{requests/proc.} & 185.8 & 48.0 & 88.6 & 218.1 & 92639 & 3127 & 23484 & - \\
\text{steals/proc.} & 56.63 & 18.47 & 26.06 & 79.25 & 18031 & 1034 & 2395 & - \\
\end{array}
\]

Table 4: Performance of Cilk on various applications. All times are in seconds, except where noted.
pfold, and ray all have critical paths under a tenth of a second long and perform fewer than 220 requests and 80 steals per processor, whereas knary(10, 5, 2) and socrates have critical paths more than 3 seconds long and perform more than 20,000 requests and 1500 steals per processor. The table does not show any clear correlation between work and either requests or steals. For example, ray does more than twice as much work as knary(10, 5, 2), yet it performs two orders of magnitude fewer requests. In Section 6, we show that for "fully strict" Cilk programs, the communication per processor grows linearly with the critical path length and does not grow as function of the work.

5 Modeling performance

In this section, we further document the effectiveness of the Cilk scheduler by showing empirically that it schedules applications in a near-optimal fashion. Specifically, we use the knary synthetic benchmark to show that the runtime of an application on P processors can be accurately modeled as $T_P \approx T_1(P) + c_\infty T_\infty$, where $c_\infty \approx 1.5$. This result shows that we obtain nearly perfect linear speedup when the critical path is short compared with the average amount of work per processor. We also show that a model of this kind is accurate even for socrates, which is our most complex application programmed to date and which does not obey all the assumptions assumed by the theoretical analyses in Section 6.

A good scheduler should try to run an application with $T_1$ work in $T_1/P$ time on $P$ processors. Such perfect linear speedup cannot be obtained whenever $T_\infty > T_1/P$, since we always have $T_P \geq T_\infty$, or more generally, $T_P \geq \max\{T_1/P, T_\infty\}$. The critical path $T_\infty$ is the stronger lower bound on $T_P$ whenever $P$ exceeds the average parallelism $T_1/T_\infty$, and $T_1/P$ is the stronger bound otherwise. A good scheduler should meet each of these bounds as closely as possible.

In order to investigate how well the Cilk scheduler meets these two lower bounds, we used our knary benchmark (described in Section 4), which can exhibit a range of values for work and critical path.

Figure 5 shows the outcome of many experiments of running knary with various values for $k$, $n$, $r$, and $P$. The figure plots the speedup $T_1/T_P$ for each run against the machine size $P$ for that run. In order to compare the outcomes for runs with different parameters, we have normalized the data by dividing the plotted values by the average parallelism $T_1/T_\infty$. Thus, the horizontal position of each datum is $P/T_1(T_\infty)$, and the vertical position of each datum is $(T_1(T_P))/(T_1(T_\infty)) = T_\infty/T_P$. Consequently, on the horizontal axis, the normalized machine-size is 1.0 when the average available parallelism is equal to the machine size. On the vertical axis, the normalized speedup is 1.0 when the runtime equals the critical path, and it is 0.1 when the runtime is 10 times the critical path. We can draw the two lower bounds on time as upper bounds on speedup. The horizontal line at 1.0 is the upper bound on speedup obtained from the critical path, and the 45-degree line is the upper bound on speedup obtained from the work per processor. As can be seen from the figure, the knary runs for which the average parallelism exceeds the number of processors (normalized machine size $< 1$), the Cilk scheduler obtains nearly perfect linear speedup. In the region where the number of processors is large compared to the average parallelism (normalized machine size $> 1$), the data is more scattered, but the speedup is always within a factor of 4 of the critical-path upper bound.

The theoretical results from Section 6 show that the expected running time of an application on $P$ processors is $T_P = O(T_1(P) + T_\infty)$. Thus, it makes sense to try to fit the data to a curve of the form $T_P = c_1(T_1(P) + c_\infty T_\infty)$. A least-squares fit to the data to minimize the relative error yields $c_1 = 0.95\pm0.1775$ and $c_\infty = 1.54\pm0.3888$ with 95 percent confidence. The $R^2$ correlation coefficient of the fit is 0.989101, and the mean relative error is 13.07 percent. The curve fit is shown in Figure 5, which also plots the simpler curves $T_P = T_1(P) + T_\infty$ and $T_P = T_1(P) + 2 \cdot T_\infty$ for comparison. As can be seen from the figure, little is lost in the linear speedup range of the curve by assuming that $c_1 = 1$. Indeed, a fit to $T_P = T_1(P) + c_\infty T_\infty$ yields $c_\infty = 1.509 \pm 0.3727$ with $R^2 = 0.983592$ and a mean relative error of 4.04 percent, which is in some ways better than the fit that includes a $c_1$ term. (The $R^2$ measure is a little worse, but the mean relative error is much better.)

It makes sense that the data points become more scattered when $P$ is close to or exceeds the average parallelism. In this range, the amount of time spent in work stealing becomes a significant fraction of the overall execution time. The real measure of the quality of a scheduler is how much larger $T_1/T_\infty$ must be than $P$ before $T_P$ shows substantial influence from the critical path. One can see from Figure 5 that if the average parallelism exceeds $P$ by a factor of 10, the critical path has almost no impact on the running time.

To confirm our simple model of the Cilk scheduler’s performance on a real application, we ran socrates on a variety of chess positions. Figure 6 shows the results of our study, which confirm the results from the knary synthetic benchmarks. The curve shown is the best fit to $T_P = c_1(T_1(P) + c_\infty T_\infty)$, where $c_1 = 1.067 \pm 0.0141$ and $c_\infty = 1.042 \pm 0.0467$ with 95 percent confidence. The $R^2$ correlation coefficient of the fit is 0.9994, and the mean relative error is 4.05 percent.

Indeed, as some of us were developing and tuning heuristics to increase the performance of socrates, we used work and critical path as our measures of progress. This methodology let us avoid being trapped by the following interesting anomaly. We made an "improvement" that sped up the program on 32 processors. From our measurements, however, we discovered that it was faster only because it saved on work at the expense of a much longer critical path. Using the simple model $T_P = T_1(P) + T_\infty$, we concluded that on a 512-processor machine, which was our platform for tournaments, the "improvement" would yield a loss of performance, a fact that we later verified. Measuring work and critical path enabled us to use experiments on a 32-processor machine to improve our program for the 512-processor machine, but without using the 512-processor machine, on which computer time was scarce.

6 A theoretical analysis of the Cilk scheduler

In this section we use algorithmic analysis techniques to prove that for the class of "fully strict" Cilk programs, Cilk’s work-stealing scheduling algorithm is efficient with respect to space, time, and communication. A fully strict program is one for which each thread sends arguments only to its parent’s successor threads. For this class of programs, we prove the following three bounds on space, time, and communication:

**Space** The space used by a $P$-processor execution is bounded by $S_P \leq S_1 P$, where $S_1$ denotes the space used by the serial execution of the Cilk program. This bound is existentially optimal to within a constant factor [3].

**Time** With $P$ processors, the expected execution time, including scheduling overhead, is bounded by $T_P = O(T_1(P) + T_\infty)$. Since both $T_1(P)$ and $T_\infty$ are lower bounds for any $P$-processor execution, our expected time bound is within a constant factor of optimal.

**Communication** The expected number of bytes communicated during a $P$-processor execution is $O(T_\infty P S_{max})$, where $S_{max}$ denotes the largest size of any closure. This bound is existentially optimal to within a constant factor [41].

The expected time bound and the expected communication bound can be converted into high-probability bounds at the cost of only a small
Figure 5: Normalized speedups for the knary synthetic benchmark using from 1 to 256 processors. The horizontal axis is $P$ and the vertical axis is the speedup $T_1 / T_p$, but each data point has been normalized by dividing the these parameters by $T_1 / T_\infty$.

Figure 6: Normalized speedups for the *Socrates chess program.
We associate each allocated closure with a primary leaf as follows. If the closure is a primary leaf, it is assigned to itself. Otherwise, if the closure has any allocated children, then it is assigned to the same primary leaf as its leftmost child. If the closure is a leaf but has some left siblings, then the closure is assigned to the same primary leaf as its leftmost sibling. In this recursive fashion, we assign every allocated closure to a primary leaf. Now, we consider the set of closures assigned to a given primary leaf. The total space of these closures is at most \( S_1 \), because this set of closures is a subset of the closures that are allocated during a 1-processor execution when the processor is executing this primary leaf, which completes the proof.

We now give the theorems bounding execution time and communication cost. Proofs for these theorems generalize the results of [3] for a more restricted model of multithreaded computation. As in [3], these proofs assume a communication model in which messages are delayed only by contention at destination processors, but no assumptions are made about the order in which contending messages are delivered [31]. The bounds given by these theorems assume that no thread has more than one successor thread.

The proofs of these theorems are analogous to the proofs of Theorems 12 and 13 in [3]. We show that certain “critical” threads are likely to be executed after only a modest number of steal requests, and that executing a critical thread guarantees progress on the critical path of the dag.

We first construct an augmented dag \( D' \) that will be used to define the critical threads. The dag \( D' \) is constructed by adding edges to the original dag \( D \) of the computation. For each child procedure \( v \) of a thread \( t \), we add an edge to \( D' \) from the first thread of \( v \) to the first thread of the next child procedure spawned by \( t \) after \( v \) is spawned. We make the technical assumption that the first thread of each procedure executes in zero time since we can add a zero-time thread to the beginning of each procedure without affecting work or depth. An example of the dag \( D' \) is given in Figure 7, where the additional edges are shown gray and curved. We draw the children spawned by a node in right-to-left order in the figure, because the execution order by the local processor is left to right, corresponding to LIFO execution. The dag \( D' \) is constructed for analytic purposes only and has no effect on the scheduling of the threads. An important property of \( D' \) is that its critical path is the same as the critical path of the original dag \( D \).

We next define the notion of a critical thread formally. We have already defined a ready thread as a thread all of whose predecessors in \( D \) have been executed. Similarly, a critical thread is a thread all of whose predecessors in \( D' \) have been executed. A critical thread must be ready, but a ready thread may or may not be critical. We now state a lemma which shows that a critical thread must be the shallowest thread in a ready queue.

**Lemma 2** During the execution of any fully strict Cilk program for which no thread has more than one successor thread, any critical thread must be the shallowest thread in a ready queue. Moreover, the critical thread is also first in the steal order.

**Proof:** For a thread \( t \) to be critical, the following conditions must hold for the ready queue on the processor in which \( t \) is enqueued:

1. No right siblings of \( t \) are in the ready queue. If a right sibling procedure \( v \) of \( t \) were in the ready queue, then the first thread of \( v \) would not have been executed, and because the first thread of \( v \) is a predecessor of \( t \) in \( D' \), \( t \) would not be critical.
2. No right siblings of any of \( t \)'s ancestors are in the ready queue. This fact follows from the same reasoning as above.
3. No left siblings of any of \( t \)'s ancestors are in the ready queue. This condition must hold because all of these siblings occur
before $t$’s parent in the local execution order, and $t$’s parent must have been executed for $t$ to be critical.

4. No successor threads of $t$’s ancestors are enabled. This condition must be true, because any successor thread must wait for all children to complete before it is enabled. Since $t$ has not completed, no successor threads of $t$’s ancestors are enabled. This condition makes use of the fact that the computation is fully strict, which implies that the only thread to which $t$ can send its result is $t$’s parent’s unique successor.

A consequence of these conditions is that no thread could possibly be above $t$ in the ready queue, because all threads above $t$ are either already executed, stolen, or not enabled. In $t$’s level, $t$ is first in the work-stealing order, because it is the rightmost thread at that level.

**Theorem 3** For any number $P$ of processors and any fully strict Cilk program in which each thread has at most one successor, if the program has work $T_1$ and critical path length $T_{\infty}$, then Cilk’s work-stealing scheduler executes the program in expected time $E[T_P] = O(T_1 \cdot P + T_{\infty})$. Furthermore, for any $\epsilon > 0$, the execution time is $T_P = O(T_1 \cdot P + T_{\infty} + \log P + \log(1/\epsilon))$ with probability at least $1 - \epsilon$.

**Proof:** This proof is just a straightforward application of the techniques in [3], using our Lemma 2 as a substitute for Lemma 9 in [3]. Because the critical threads are first in the work-stealing order, they are likely to be stolen (or executed locally) after a modest number of steal requests. This fact can be shown formally using a delay sequence argument.

**Theorem 4** For any number $P$ of processors and any fully strict Cilk program in which each thread has at most one successor, if the program has critical path length $T_{\infty}$ and maximum closure size $S_{\text{max}}$, then Cilk’s work-stealing scheduler incurs expected communication $O((T_{\infty} \cdot P \cdot S_{\text{max}})$ Furthermore, for any $\epsilon > 0$, the communication cost is $O((T_{\infty} + \log(1/\epsilon)) \cdot P \cdot S_{\text{max}})$ with probability at least $1 - \epsilon$.

**Proof:** This proof is exactly analogous to the proof of Theorem 13 in [3]. We observe that at most $O(T_{\infty} \cdot P)$ steal attempts occur in an execution, and all communication costs can be associated with one of these steal requests such that at most $O(S_{\text{max}})$ communication is associated with each steal request. The high-probability bound is analogous.

7 Conclusion

To produce high-performance parallel applications, programmers often focus on communication costs and execution time, quantities that are dependent on specific machine configurations. We argue that a programmer should think instead about work and critical path, 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 are observable quantities, and it delivers guaranteed performance as a function of these quantities. Work and critical path have been used in the theory community for years to analyze parallel algorithms [26]. Blelloch [2] has developed a performance model for data-parallel computations based on these same two abstract measures. He cites many advantages to such a model over machine-based models. Cilk provides a similar performance model for the domain of asynchronous, multithreaded computation.

Although Cilk offers performance guarantees, its current capabilities are limited, and programmers find its explicit continuation-passing style to be onerous. Cilk is good at expressing and executing dynamic, asynchronous, tree-like, MIMD computations, but it is not yet ideal for more traditional parallel applications that can be programmed effectively in, for example, a message-passing, data-parallel, or single-threaded, shared-memory style. We are currently working on extending Cilk’s capabilities to broaden its applicability. A major constraint is that we do not want new features to destroy Cilk’s guarantees of performance. Our current research focuses on implementing “dag-consistent” shared memory, which allows programs to operate on shared memory without costly communication or hardware support; on providing a linguistic interface that produces continuation-passing code for our runtime system from a more traditional call-return specification of spawns; and on incorporating persistent threads and less strict semantics in ways that do not destroy the guaranteed performance of our scheduler. Recent information about Cilk is maintained on the World Wide Web in page http://theory.lcs.mit.edu/~cilk.

Acknowledgments

We gratefully acknowledge the inspiration of Michael Halbher, now of the Boston Consulting Group in Zurich, Switzerland. Mike’s PCM runtime system [18] developed at MIT was the precursor of Cilk, and many of the design decisions in Cilk are owed to him. We thank Shail Aditya and Sivan Toledo of MIT and Larry Rudolph of Hebrew University for helpful discussions. Xinnin Tian of McGill University provided helpful suggestions for improving the paper. Rolf Riesen of Sandia National Laboratories ported Cilk to the Intel Paragon MPP running under the SUNMOS operating system, John Litvin and Mike Stupak ported Cilk to the Paragon running under OSF, and Andy Shaw of MIT ported Cilk to the Silicon Graphics Power Challenge SMP. Thanks to Matteo Frigo and Rob Miller of MIT for their many contributions to the Cilk system. Thanks to the Scout project at MIT and the National Center for Supercomputing Applications at University of Illinois, Urbana-Champaign for access to their CM5 supercomputers for running our experiments. Finally, we acknowledge the influence of Arvind and his dataflow research group at MIT. Their pioneering work attracted us to this path, and their vision continues to draw us forward.

References


