# Cilk: An Efficient Multithreaded Runtime System

Robert Blumofe
MIT Laboratory for Computer Science
Cambridge, MA 02139
rdb@theory.lcs.mit.edu

Multithreading has become an increasingly popular way to implement dynamic, highly asynchronous, concurrent programs. A multithreaded system provides the programmer with a means to create, synchronize, and schedule threads. Although the schedulers in many runtime systems seem to perform well in practice, none provide users with a guarantee of application performance. Cilk (pronounced "silk") 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.

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 network of workstations. To date, the applications we have programmed include protein folding, graphic rendering, backtrack search, and the $\star$Socrates chess program, which won third prize in the 1994 ACM International Computer Chess Championship.

A Cilk multithreaded computation can be viewed as a directed acyclic graph (dag) that unfolds dynamically. A Cilk program consists of a collection of Cilk *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 a thread runs, it can *spawn* a child thread. 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. 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.

This research was supervised by Professor Charles E. Leiserson and was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985 and Project SCOUT (contract MDA97292-J-1032), and by an ARPA High-Performance Computing Graduate Fellowship.

Other members of the Cilk team, present and past, include Mateo Frigo, Michael Halbherr, Chris Joerg, Bradley Kuszmaul, Rob Miller, Keith Randall, and Yuli Zhou

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 $T_1$, 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 $T_\infty$, 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 in the dag. With $P$ processors, the execution time $T_P$ cannot be less than $T_1/P$ (perfect linear speedup) or less than $T_\infty$. A good scheduler should meet each of these lower bounds as closely as possible.

The Cilk scheduler uses random "work stealing" [2] to achieve execution time very near to the sum of these two lower bounds. Off-line techniques for computing such efficient schedules have been known for a long time, but this efficiency has been difficult to achieve on-line in a distributed environment while simultaneously using small amounts of space and communication.

In [1], we used algorithmic and empirical techniques to show that for the class of "fully strict" programs, Cilk's work-stealing scheduler 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. (All of the applications we have coded to date are fully strict.) Analytically, we showed that for any number $P$ of processors and any fully strict computation with work $T_1$ and critical path $T_\infty$, the Cilk scheduling algorithm achieves execution time $T_P = O(T_1/P + T_\infty)$ with high probability while simultaneously using amounts of space and communication that are existentially optimal to within a small constant factor. The remainder of this paper focuses on empirical measurements of execution time on the CM5 for our $\star$Socrates chess application.

Figure 1 shows the outcome of many experiments of running $\star$Socrates on a variety of chess positions. 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 parallelism is equal to the machine size. On the ver-

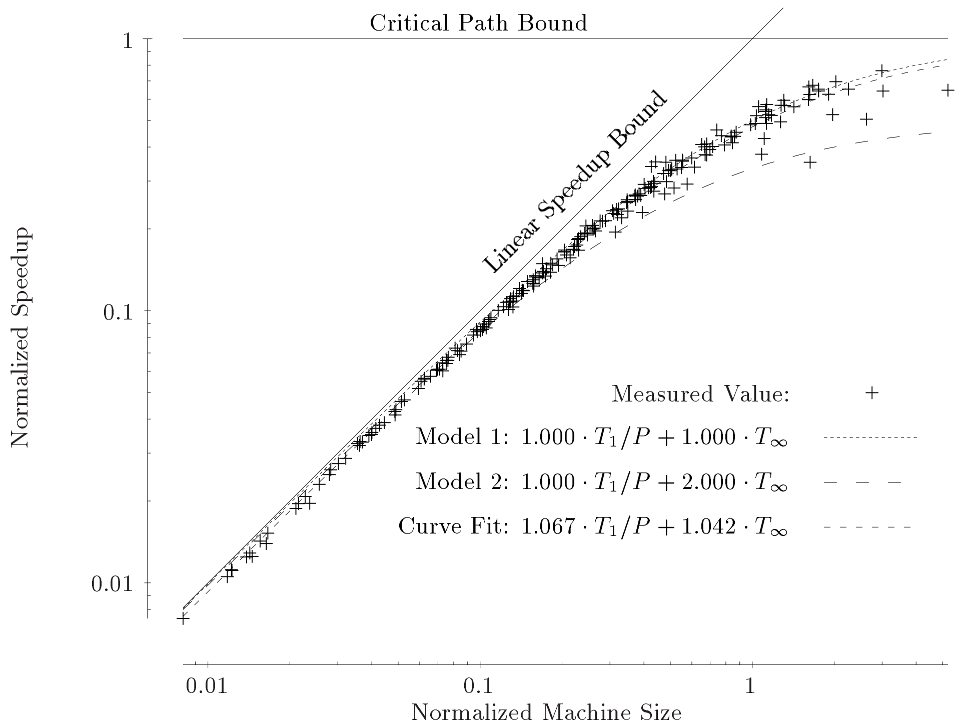


**Figure 1**: Normalized speedups for the ⋆Socrates chess program.

tical axis, the normalized speedup is $1.0$ when the runtime equals the critical path. As can be seen from the figure, on the ⋆Socrates 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 [2] 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 $c_1 = 1.067 \pm 0.0141$ 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. The curve fit is shown in Figure 1, 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. This curve fit provides an accurate model of application performance based on work and critical path.

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.

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 characterize an algorithm independent of 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.

Although Cilk offers performance guarantees, its current capabilities are limited, and programmers find its explicit continuation-passing style to be onerous. Our current research focuses on implementing "dag-consistent" shared memory, and on providing a linguistic interface that supports a more traditional call-return specification of spawns. Recent information about Cilk is maintained on the World Wide Web in page `http://theory.lcs.mit.edu/~cilk`.

# References


[1] 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)*, Santa Barbara, California, July 1995. To appear.

[2] 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.