Cilk: An Efficient Multithreaded Runtime System*

ROBERT D. BLUMOFE, CHRISTOPHER F. JOERG, BRADLEY C. KUSZMAUL, CHARLES E. LEISERSON, KEITH H. RANDALL, AND YULI ZHOU

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

October 17, 1996

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 length" of a Cilk computation can be used to model performance accurately. Consequently, a Cilk programmer can focus on reducing the computation's work and critical-path length, 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 Sun Sparcstation SMP, and the Cilk-NOW network of workstations. Applications written in Cilk include protein folding, graphic rendering, backtrack search, and the $\star$Socrates chess program, which won second prize in the 1995 ICCA World Computer Chess Championship.

1 Introduction

Multithreading has become an increasingly popular way to implement dynamic, highly asynchronous, concurrent programs [1, 9, 10, 11, 12, 13, 16, 21, 23, 24, 26, 27, 30, 36, 37, 39, 42, 43].

*This research was supported in part by the Advanced Research Projects Agency under Grants N00014-94-1-0985 and N00014-92-J-1310. Robert Blumofe was supported in part by an ARPA High-Performance Computing Graduate Fellowship. Keith Randall is supported in part by a Department of Defense NDSEG Fellowship.

Robert D. Blumofe's current address is The University of Texas at Austin; Department of Computer Sciences; Taylor Hall; Austin, TX 78712. Christopher F. Joerg's current address is Digital Equipment Corporation; Cambridge Research Laboratory; One Kendall Square, Building 700, 2nd Floor; Cambridge, MA 02139. Bradley C. Kuszmaul's current address is Yale University; Department of Computer Science; 51 Prospect Street; New Haven, CT 06520. Yuli Zhou's current address is AT&T Bell Laboratories; 600 Mountain Avenue; Murray Hill, NJ 07974.

Published in The Journal of Parallel and Distributed Computing, 37(1):55-69, Aug. 25 1996.

Portions of this work were previously reported in the Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara, CA, July 19–21, 1995, pp. 207–216.

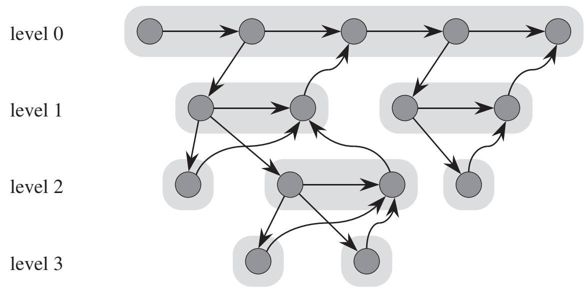


Figure 1: The Cilk model of multithreaded computation. Threads are shown as circles, which are grouped into procedures. Each downward edge corresponds to a spawn of a child, each horizontal edge corresponds to a spawn of a successor, and each upward, curved edge corresponds to a data dependency. The levels of procedures in the spawn tree are indicated at the left.

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 length" 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 is 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 must 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 length*. 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_∞, is the total amount of time required by an ideal infinite-processor execution, which corresponds to the largest sum of thread execution times along any path. With P processors, the execution time cannot be less than T_1/P or less than T_∞. The Cilk scheduler uses "work stealing" [4, 8, 14, 15, 16, 21, 29, 30, 31, 37, 43] 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 [6, 18, 19], 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 second prize in the 1995 World Computer Chess Championship running on Sandia National Laboratories' 1824-node Intel Paragon MPP. Many of these applications pose problems for more traditional parallel environments, such as message passing [41] and data parallel [2, 22], 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 extends C with primitives to express parallelism, and the Cilk runtime system maps the expressed parallelism into a parallel execution. A Cilk program is preprocessed to C using our `cilk2c` type-checking preprocessor [34]. The C code is then compiled and linked with a runtime library to run on the target platform. Currently supported targets include the Connection Machine CM5 MPP, the Intel Paragon MPP, the Sun Sparcstation SMP, and the Cilk-NOW [3, 5] 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 40 pages of C, and it performs communication among processors using the Strata [7] active-message library. This description of the Cilk environment corresponds to the version as of March 1995.

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 length of a Cilk computation can be used to model performance. Section 6 shows analytically that the scheduler works well. Finally, Section 7 offers some concluding remarks and describes our plans for the future.

2 The Cilk programming environment and implementation

This section describes the basic runtime data structures that Cilk uses, as well as a C language extension that we have developed to ease the task of coding Cilk programs. The runtime system is based on a data structure, called a "closure," which is manipulated in a very simple fashion by the language. The Cilk "language" is less a higher-level language than it is an assembly language for multithreading. It employs "explicit continuation passing" to connect lightweight threads together into procedures. We do not view this primitive Cilk language as a research contribution, but rather as a tool to simplify exploitation of the multithreaded runtime system.

A Cilk procedure is broken into a sequence of threads. The procedure itself is not explicitly specified in the language. Rather, it is defined implicitly in terms of its constituent threads. A thread *T* is defined in a manner similar to a C function definition:

```
thread T (arg-decls ...)  { stmts ...}
```

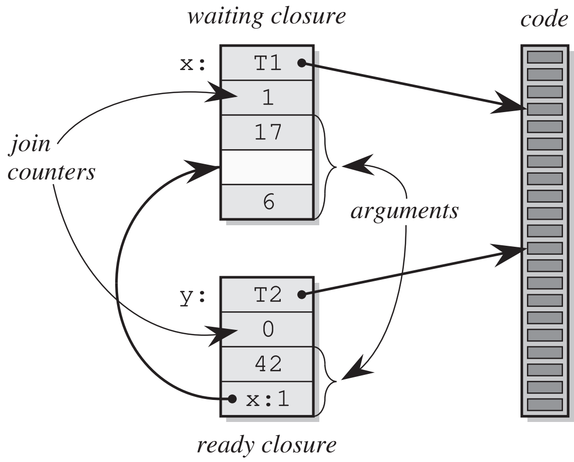


Figure 2: The closure data structure.

The `cilk2c` type-checking preprocessor [34] 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 C function 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 `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, and hence a child procedure, by creating a closure for the child. Spawning is specified in the Cilk language as follows:

```
spawn T (args ...)
```

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 with ordinary C syntax. To specify a missing argument, the user specifies a continuation variable (type `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 `spawn` causes the closure to be immediately posted to the scheduler for execution. In typical applications, child closures are created with no missing arguments.

To string a sequence of threads together to make a Cilk procedure, each thread in the procedure is responsible for creating its successor. To create a successor thread, a thread executes the following statement:

```
spawn_next T (args ...)
```

This statement is semantically identical to `spawn`, but it informs the scheduler that the new closure should be treated as a successor, as opposed to as 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:

```
send_argument (k, value)
```

This statement sends the value `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 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 [21, 27, 35, 37] 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 that the runtime system employs either multiple stacks or a mechanism to save these temporaries in heap-allocated storage. 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.

```
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 nth Fibonacci number.

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 `tail_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.

Cilk provides an "assembly language" for programming multithreaded systems, which means that programmers can access and exploit the power of multithreaded programming, but they must deal with the many details imposed by explicit continuation passing. Programming in Cilk can be difficult, and we do not view the current Cilk language as being a desirable programming interface for multithreading. Our research has focused on developing a provably good runtime system, however, rather than on language design. Recognizing the linguistic deficiencies of Cilk as described in this paper, we have recently been developing a version of the system that does not use explicit continuation passing, has a cleaner language, yet still provides performance guarantees.

3 The Cilk work-stealing scheduler

Cilk's scheduler uses the technique of *work stealing* [4, 8, 14, 15, 16, 21, 29, 30, 31, 37, 43] 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 [4, 29, 40]. We shall now present the implementation of Cilk's work-stealing scheduler.

At runtime, each processor maintains a local *ready pool* to hold ready closures. Each closure has an associated *level*, which corresponds to the thread's depth in the spawn tree. The closures for the threads in the root procedure have level 0, the closures for the threads in the root's child procedures have level 1, and so on. The ready pool is an array, illustrated in Figure 4, in which the

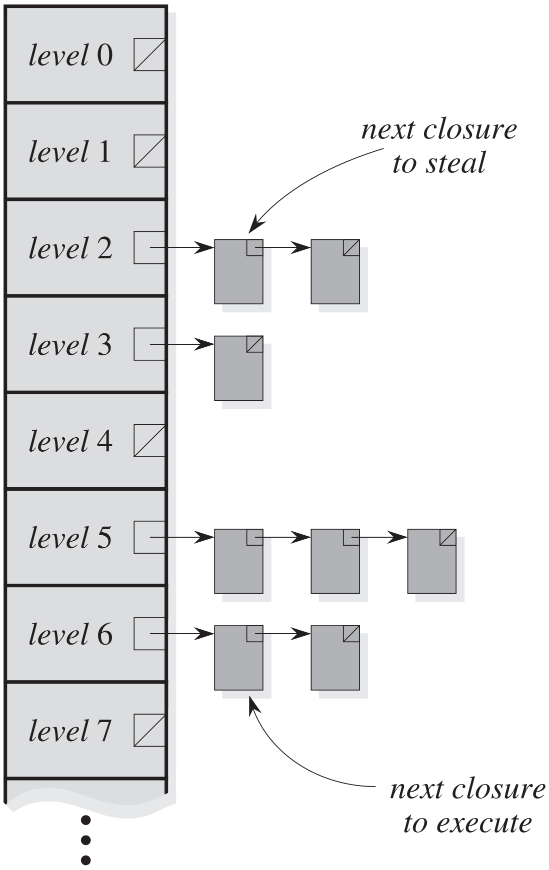


Figure 4: A processor's ready pool. At each iteration through the scheduling loop, the processor executes the closure at the head of the deepest nonempty level in the ready pool. If the ready pool is empty, the processor becomes a thief and steals the closure at the head of the shallowest nonempty level in the ready pool of a victim processor chosen uniformly at random.

Lth element contains a linked list of all ready closures having level L.

Cilk begins executing the user program by initializing all ready pools to be empty, placing the initial root thread into the level-0 list of Processor 0's pool, and then starting a scheduling loop on each processor.

At each iteration through the scheduling loop, a processor first checks to see whether its ready pool 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 closure at the head of the list of the deepest nonempty level in the ready pool.
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 dies, control returns to the scheduling loop which advances to the next iteration.

When a thread at level L performs a `spawn` of a child thread `T`, the processor 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 pool by inserting it at the head of the level-$(L+1)$ list.

Execution of `spawn_next` is similar, except that the closure is labeled with level L and, if it is ready, posted to the level-L list.

When a thread performs a `send_argument (k, value)`, the processor executes the following operations:

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 pool 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 pool of the initiating processor, rather than to that of the remote processor. This policy is necessary for the scheduler to be provably efficient, but as a practical matter, we have also had success with posting the closure to the remote processor's pool.

If a processor begins an iteration of the scheduling loop and finds that its ready pool is empty, 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 pool is empty, go back to Step 1.
3. If the victim's ready pool is nonempty, extract the closure from the head of the list in the shallowest nonempty level of the ready pool, and execute it.

Work stealing is implemented with a simple request-reply communication protocol between the thief and victim.

Why steal work from the shallowest level of the ready pool? The reason is two-fold—one heuristic and one algorithmic. First, to lower communication costs, we would like to steal large amounts of work, and in a tree-structured computation, shallow threads are likely to spawn more work than deep ones. This heuristic notion is the justification cited by earlier researchers [8, 15, 21, 35, 43] who proposed stealing work that is shallow in the spawn tree. We cannot, however, prove that shallow threads are more likely to spawn work than deep ones. What we prove in Section 6 is the following algorithmic property. The threads that are on the "critical path" in the dag, are always at the shallowest level of a processor's ready pool. Consequently, if processors are idle, the work they steal makes progress along the critical path.

4 Performance of Cilk applications

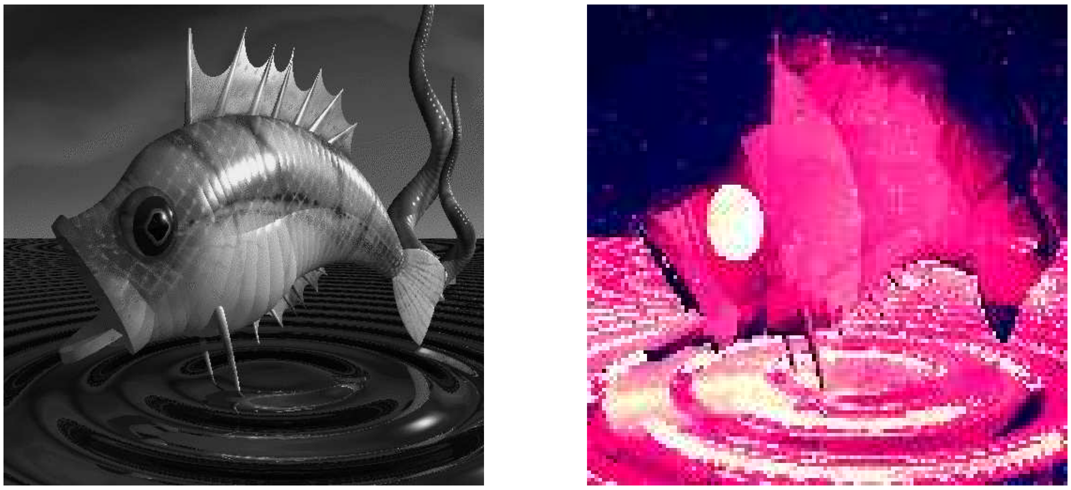
The Cilk runtime system executes Cilk applications efficiently and with predictable performance. Specifically, for dynamic, asynchronous, tree-like applications, Cilk's work-stealing scheduler produces near optimal parallel speedup while using small amounts of space and communication. Furthermore, Cilk application performance can be modeled accurately as a simple function of work and critical-path length. In this section, we empirically demonstrate these facts by experimenting with several applications. This section begins with a look at these applications and then proceeds with a look at the performance of these applications. In the next section we look at application performance modeling. The empirical results of this section and the next confirm the analytical results of Section 6.

Cilk applications

We experimented with the Cilk runtime system using several applications, some synthetic and some real. The applications are described below:

- `fib(n)` 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(n)` 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 7 levels of the search tree.
- `pfold(x,y,z)` is a protein-folding program that finds hamiltonian paths in a three-dimensional grid of size $x \times y \times z$ using backtrack search [38]. Written by Chris Joerg of MIT's Laboratory for Computer Science and V. Pande of MIT's Center for Material Sciences and Engineering, `pfold` was the first program to enumerate all hamiltonian paths in a $3 \times 4 \times 4$ grid. For our experiments, we timed the enumeration of all paths starting with a certain sequence.
- `ray(x,y)` 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 of size $x \times y$. 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(n,k,r)` is a synthetic benchmark whose parameters can be set to produce a variety of values for work and critical-path length. It generates a tree of depth `n` and branching factor `k` 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.
- $\star$Socrates is a parallel chess program that uses the Jamboree search algorithm [25, 31] to parallelize a minmax tree search. The work of the algorithm varies with the number of processors, because it does speculative work that may be aborted during runtime. $\star$Socrates won second prize in the 1995 ICCA World Computer Chess Championship running on the 1824-node Intel Paragon at Sandia National Laboratories.

[1] Initially, the Cilk `ray` program was about 5 percent faster than the serial `POV-Ray` program 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 Figure 6.



(a) Ray-traced image.

(b) Work at each pixel.

Figure 5: (a) An image rendered with the `ray` program. (b) This image shows the amount of time `ray` took to compute each pixel value. The whiter the pixel, the longer `ray` worked to compute the corresponding pixel value.

Many of these applications place heavy demands on the runtime system due to their dynamic and irregular nature. For example, in the case of `queens` and `pfold`, the size and shape of the backtrack-search tree cannot be determined without actually performing the search, and the shape of the tree often turns out to be highly irregular. With speculative work that may be aborted, the ⋆Socrates minmax tree carries this dynamic and irregular structure to the extreme. In the case of `ray`, the amount of time it takes to compute the color of a pixel in an image is hard to predict and may vary widely from pixel to pixel, as illustrated in Figure 5. In all of these cases, high performance demands efficient, dynamic load balancing at runtime.

All experiments were run on a CM5 supercomputer. The CM5 is a massively parallel computer based on 32MHz SPARC processors with a fat-tree interconnection network [32]. The Cilk runtime system on the CM5 performs communication among processors using the Strata [7] active-message library.

Application performance

By running our applications and measuring a suite of performance parameters, we empirically answer many questions about the effectiveness of the Cilk runtime system. We focus on the following questions. How efficiently does the runtime system implement the language primitives? As we add processors, how much faster will the program run? How much more space will it require? How much more communication will it perform? We show that for dynamic, asynchronous, tree-like programs, the Cilk runtime system efficiently implements the language primitives, and that it is simultaneously efficient with respect to time, space, and communication. In Section 6, we reach the same conclusion by analytic means, but in this section we focus on empirical data from the execution of our Cilk programs.

The execution of a Cilk program with a given set of inputs grows a *Cilk computation* that consists of a tree of procedures and a dag of threads. These structures were introduced in Section 1. We benchmark our applications with respect to work and critical-path length.

Recall that the work, denoted by T_1, is the time to execute the Cilk computation on one processor, which corresponds to the sum of the execution times of all the threads in the dag. The method used to measure T_1 depends on whether the program is deterministic. For deterministic programs, the computation only depends on the program and its inputs, and hence, it is independent of the number of processors and runtime scheduling decisions.[2] All of our applications, except $\star$Socrates, are deterministic. For these deterministic applications, the work performed by any P-processor run of the program is equal to the work performed by a 1-processor run (with the same input values), so we measure the work T_1 directly by timing the 1-processor run. The $\star$Socrates program, on the other hand, uses speculative execution, and therefore, the computation depends on the number of processors and scheduling decisions made at runtime. In this case, timing a 1-processor run is not a reasonable way to measure the work performed by a run with more processors. We must realize that the work T_1 of an execution with P processors is defined as the time it takes 1-processor to execute the same *computation*, not the same program (with the same inputs). For $\star$Socrates we estimate the work of a P-processor run by performing the P-processor run and timing the execution of every thread and summing. This method yields an underestimate, since it does not include scheduling costs. In either case, a P-processor execution of a Cilk computation with work T_1 must take time at least T_1/P.[3] A P-processor execution that takes time equal to this T_1/P lower bound is said to achieve *perfect linear speedup*.

Recall that the critical-path length, denoted by T_∞, is the time to execute the Cilk computation with infinitely many processors, which corresponds to the largest sum of thread execution times along any path in the dag. Cilk can measure critical-path length by timestamping each thread in the dag with the earliest time at which it could have been executed. Specifically this timestamp is the maximum of the earliest time that the thread could have been spawned and, for each argument, the earliest time that the argument could have been sent. These values, in turn, are computed from the timestamp of the thread that performed the spawn or sent the argument. In particular, if a thread performs a spawn, then the earliest time that the spawn could occur is equal to the earliest time at which the thread could have been executed (its timestamp) plus the amount of time the thread ran for until it performed the spawn. The same property holds for the earliest time that an argument could be sent. The initial thread of the computation is timestamped zero, and the critical-path length is then computed as the maximum over all threads of its timestamp plus the amount of time it executes for. The measured critical-path length does not include scheduling and communication costs. A P-processor execution of a Cilk computation must take at least as long as the computation's critical-path length T_∞. Thus, if T_∞ exceeds T_1/P, then perfect linear speedup cannot be achieved.

Figure 6 is a table showing typical performance measures for our Cilk applications. Each column presents data from a single run of a benchmark application. We adopt the following notations, which are used in the table. For each application, we have an efficient serial C implementation, compiled using `gcc -O2`, whose measured runtime is denoted T_{serial}. The Cilk computation's

[2]Randomized programs can be viewed as deterministic if we consider the sequence of values generated by the source of randomness to be inputs to the program.

[3]In practice, we sometimes beat the T_1/P lower bound. Such *superlinear speedup* is a consequence of the fact that as we add processors, we also add other physical resources such as registers, cache, and main memory.

	fib (33)	queens (15)	pfold (3,3,4)	ray (500,500)	knary (10,5,2)	knary (10,4,1)	⋆Socrates (depth 10) (32 proc.)	⋆Socrates (depth 10) (256 proc)
(computation parameters)								
T_{serial}	8.487	252.1	615.15	729.2	288.6	40.993	1665	1665
T_1	73.16	254.6	647.8	732.5	314.6	45.43	3644	7023
T_{serial}/T_1	0.116	0.9902	0.9496	0.9955	0.9174	0.9023	0.4569	0.2371
T_∞	0.000326	0.0345	0.04354	0.0415	4.458	0.255	3.134	3.24
T_1/T_∞	224417	7380	14879	17650	70.56	178.2	1163	2168
threads	17,108,660	210,740	9,515,098	424,475	5,859,374	873,812	26,151,774	51,685,823
thread length	4.276μs	1208μs	68.08μs	1726μs	53.69μs	51.99μs	139.3μs	135.9μs
(32-processor experiments)								
T_P	2.298	8.012	20.26	21.68	15.13	1.633	126.1	-
$T_1/P + T_\infty$	2.287	7.991	20.29	22.93	14.28	1.675	117.0	-
T_1/T_P	31.84	31.78	31.97	33.79	20.78	27.81	28.90	-
$T_1/(P \cdot T_P)$	0.9951	0.9930	0.9992	1.0558	0.6495	0.8692	0.9030	-
space/proc.	70	95	47	39	41	42	386	-
requests/proc.	185.8	48.0	88.6	218.1	92639	3127	23484	-
steals/proc.	56.63	18.47	26.06	79.25	18031	1034	2395	-
(256-processor experiments)								
T_P	0.2892	1.045	2.590	2.765	8.590	0.4636	-	34.32
$T_1/P + T_\infty$	0.2861	1.029	2.574	2.903	5.687	0.4325	-	30.67
T_1/T_P	253.0	243.7	250.1	265.0	36.62	98.00	-	204.6
$T_1/(P \cdot T_P)$	0.9882	0.9519	0.9771	1.035	0.1431	0.3828	-	0.7993
space/proc.	66	76	47	32	48	40	-	405
requests/proc.	73.66	80.40	97.79	82.75	151803	7527	-	30646
steals/proc.	24.10	21.20	23.05	18.34	6378	550	-	1540

Figure 6: Performance of Cilk on various applications. All times are in seconds, except where noted.

work T_1 and critical-path length T_∞ are measured on the CM5 as described above. The measured execution time of the Cilk program running on P processors of the CM5 is given by T_P. The row labeled "threads" indicates the number of threads executed, and "thread length" is the average thread length (work divided by the number of threads).

Certain derived parameters are also displayed in the table. The ratio T_{serial}/T_1 is the *efficiency* of the Cilk program relative to the C program. The ratio T_1/T_∞ is the *average parallelism*. The value $T_1/P + T_\infty$ is a simple model of the runtime, which will be discussed later. The *speedup* is T_1/T_P, and the *parallel efficiency* is $T_1/(P \cdot T_P)$. The row labeled "space/proc." indicates the maximum number of closures allocated at any time on any processor. The row labeled "requests/proc." indicates the average number of steal requests made by a processor during the execution, and "steals/proc." gives the average number of closures actually stolen.

The data in Figure 6 shows two important relationships: one between efficiency and thread length, and another between speedup and average parallelism.

Considering the relationship between efficiency T_{serial}/T_1 and thread length, we see that for programs with moderately long threads, the Cilk runtime system induces little overhead. The `queens`, `pfold`, `ray`, and `knary` programs have threads with average length greater than 50 microseconds and have efficiency greater than 90 percent. On the other hand, the `fib` program has low efficiency, because the threads are so short: `fib` does almost nothing besides `spawn` and `send_argument`.

Despite it's long threads, the $\star$Socrates program has low efficiency, because its parallel Jamboree search algorithm is based on speculatively searching subtrees that are not searched by a serial algorithm. Consequently, as we increase the number of processors, the program executes more threads and, hence, does more work. For example, the 256-processor execution did 7023 seconds of work whereas the 32-processor execution did only 3644 seconds of work. Both of these executions did considerably more work than the serial program's 1665 seconds of work. Thus, although we observe low efficiency, it is due to the parallel algorithm and not to Cilk overhead.

Looking at the speedup T_1/T_P measured on 32 and 256 processors, we see that when the average parallelism T_1/T_∞ is large compared with the number P of processors, Cilk programs achieve nearly perfect linear speedup, but when the average parallelism is small, the speedup is much less. The `fib`, `queens`, `pfold`, and `ray` programs, 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.[4] The $\star$Socrates program exhibits somewhat less parallelism and also somewhat less speedup. On 32 processors the $\star$Socrates program has 1163-fold parallelism, yielding 90 percent of perfect linear speedup, while on 256 processors it has 2168-fold parallelism yielding 80 percent of perfect linear speedup. With even less parallelism, as exhibited in the `knary` benchmarks, less speedup is obtained. For example, the `knary(10,5,2)` benchmark exhibits only 70-fold parallelism, and it realizes barely more than 20-fold speedup on 32 processors (less than 65 percent of perfect linear speedup). With 178-fold parallelism, `knary(10,4,1)` achieves 27-fold speedup on 32 processors (87 percent of perfect linear speedup), but only 98-fold speedup on 256 processors (38 percent of perfect linear speedup).

Although these speedup measures reflect the Cilk scheduler's ability to exploit parallelism, to obtain *application speedup*, we must factor in the efficiency of the Cilk program compared

[4]In fact, the `ray` program achieves superlinear speedup even when comparing to the efficient serial implementation. We suspect that cache effects cause this phenomenon.

with the serial C program. Specifically, the application speedup T_{serial}/T_P is the product of efficiency T_{serial}/T_1 and speedup T_1/T_P. For example, applications such as fib and $\star$Socrates with low efficiency generate correspondingly low application speedup. The $\star$Socrates program, with efficiency 0.2371 and speedup 204.6 on 256 processors, exhibits application speedup of $0.2371 \cdot 204.6 = 48.51$. For the purpose of understanding scheduler performance, we have decoupled the efficiency of the application from the efficiency of the scheduler.

Looking more carefully at the cost of a spawn in Cilk, we find that it takes a fixed overhead of about 50 cycles to allocate and initialize a closure, plus about 8 cycles for each word argument. In comparison, a C function call on a CM5 SPARC processor takes 2 cycles of fixed overhead (assuming no register window overflow) plus 1 cycle for each word argument (assuming all arguments are transferred in registers). Thus, a spawn in Cilk is roughly an order of magnitude more expensive than a C function call. This Cilk overhead is quite apparent in the fib program, which does almost nothing besides spawn and send_argument. Based on fib's measured efficiency of 0.116, we can conclude that the aggregate average cost of a spawn/send_argument 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 Figure 6, 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 [17, 27, 35] have developed techniques for implementing spawns such that when the child thread executes on the same processor as its parent, the cost of the spawn operation is roughly equal the cost of a 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 Figure 6.

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, $\star$Socrates on 32 processors executes over 26 million threads, yet no processor ever contains more than 386 allocated closures. 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 an important class of Cilk programs, the space per processor does not grow as we add processors.

Looking at the "requests/proc." and "steals/proc." rows in Figure 6, we observe that the amount of communication grows with the critical-path length but does not grow with the work. For example, fib, queens, pfold, and ray all have critical-path lengths under a tenth of a second long and perform fewer than 220 requests and 80 steals per processor, whereas knary(10,5,2) and $\star$Socrates have critical-path lengths 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 a class of Cilk programs, the communication per processor grows at most linearly with the critical-path length and does not grow as a function of the work.

5 Modeling performance

We further document the effectiveness of the Cilk scheduler by showing empirically that Cilk application performance can be modeled accurately with a simple function of work T_1 and critical-path length T_∞. Specifically, we use the `knary` synthetic benchmark to show that the runtime of an application on P processors can be modeled as $T_P \approx T_1/P + c_\infty T_\infty$, where c_∞ is a small constant (about 1.5 for `knary`) determined by curve fitting. 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.

We would like our scheduler to execute a Cilk computation with T_1 work in T_1/P time on P processors. Such perfect linear speedup cannot be obtained whenever the computation's critical-path length T_∞ exceeds 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 length T_∞ is the stronger lower bound on T_P whenever P exceeds the average parallelism T_1/T_∞, 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 synthetic `knary` benchmark, which can grow computations that exhibit a range of values for work and critical-path length.

Figure 7 shows the outcome from many experiments of running `knary` with various input values (`n`, `k`, and `r`) on various numbers of processors. The figure plots the measured 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 input values, we have normalized the plotted value for each run as follows. In addition to the speedup, we measure for each run the work T_1 and the critical-path length T_∞, as previously described. We then normalize the machine size and the speedup by dividing these values by the average parallelism T_1/T_∞. For each run, the horizontal position of the plotted datum is $P/(T_1/T_\infty)$, and the vertical position of the plotted 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 number of processors is equal to the average parallelism. On the vertical axis, the normalized speedup is 1.0 when the runtime equals the critical-path length. 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 length, $T_P \geq T_\infty$, and the 45-degree line is the linear speedup bound, $T_P \geq T_1/P$. As can be seen from the figure, on the `knary` runs for which the average parallelism exceeds the number of processors (normalized machine size less than 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 greater than 1), the data is more scattered, but the speedup is always within a factor of 4 of the critical-path length upper bound.

The theoretical results from Section 6 show that the expected running time of a Cilk computation on P processors is $T_P = O(T_1/P + T_\infty)$. Thus, it makes sense to try to fit the `knary` 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 $c_1 = 0.9543 \pm 0.1775$ and $c_\infty = 1.54 \pm 0.3888$ 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 7, 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 the coefficient c_1 on the T_1/P term equals 1. Indeed,

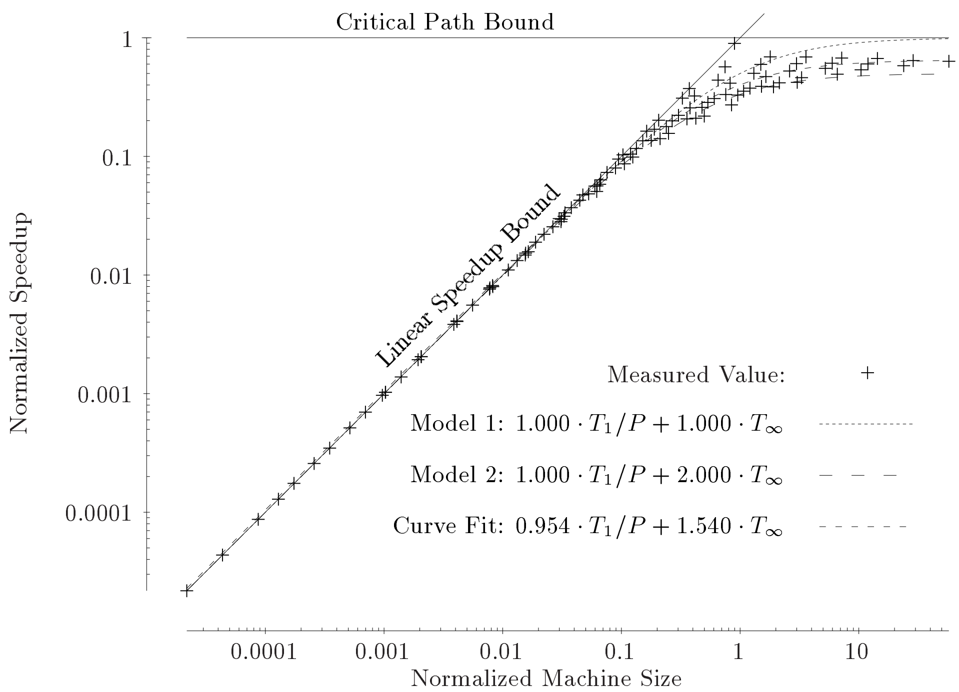


Figure 7: Normalized speedups for the `knary` synthetic benchmark using from 1 to 256 processors. The horizontal axis is the number P of processors and the vertical axis is the speedup T_1/T_P, but each data point has been normalized by dividing by T_1/T_∞.

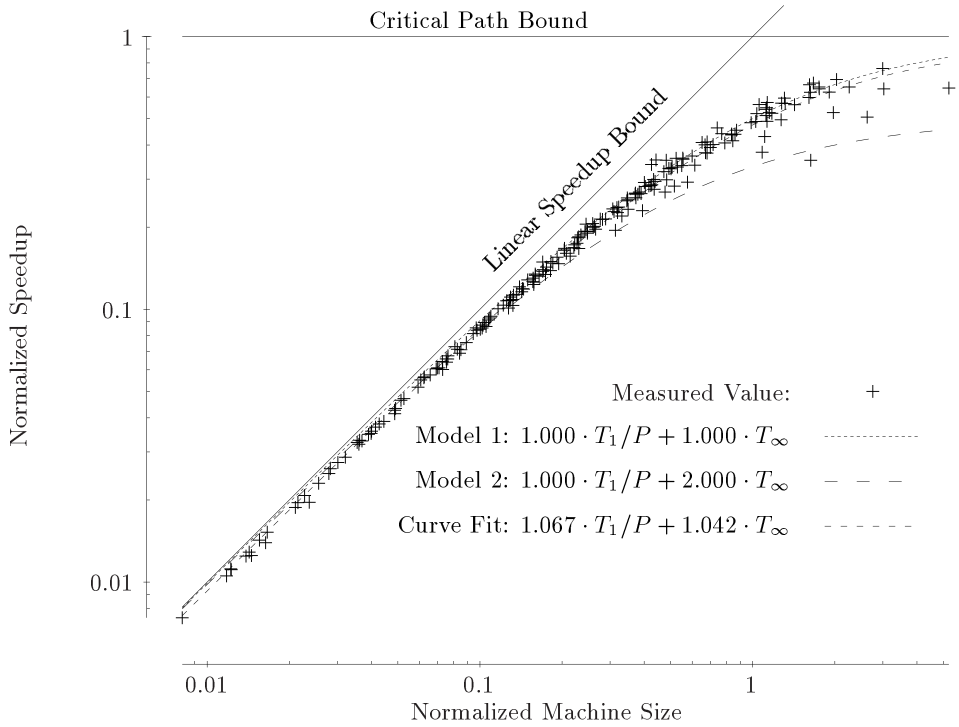


Figure 8: Normalized speedups for the ⋆Socrates chess program.

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 than P the average parallelism T_1/T_∞ must be before T_P shows substantial influence from the critical-path length. One can see from Figure 7 that if the average parallelism exceeds P by a factor of 10, the critical-path length 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 $\star$Socrates on a variety of chess positions using various numbers of processors. Figure 8 shows the results of our study, which confirm the results from the `knary` synthetic benchmark. The best fit to $T_P = c_1(T_1/P) + c_\infty(T_\infty)$ 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.

Indeed, as some of us were developing and tuning heuristics to increase the performance of $\star$Socrates, we used work and critical-path length 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 Connection Machine CM5 MPP at the National Center for Supercomputer Applications at the University of Illinois, Urbana-Champaign, which was our platform for our early tournaments, the "improvement" would yield a loss of performance, a fact which we later verified. Measuring work and critical-path length 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. In the analysis and bounds of this section, we further assume that each thread spawns at most one successor thread. Programs such as $\star$Socrates violate this assumption, and at the end of the section, we explain how the analysis and bounds can be generalized to handle such programs. For fully strict programs, we prove the following three bounds on space, time, and communication:

Space The space used by a P-processor execution is bounded by $S_P \le S_1 P$, 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 [4].

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

Communication The expected number of bytes communicated during a P-processor execution is $O(PT_\infty S_{\text{max}})$, where S_{max} is the size of the largest closure in the computation. This bound is existentially optimal to within a constant factor [44].

The expected-time bound and the expected-communication bound can be converted into high-probability bounds at the cost of only a small additive term in both cases. Full proofs of these bounds, using generalizations of the techniques developed in [4], can be found in [3].

The space bound can be obtained from a "busy-leaves" property [4] that characterizes the allocated closures at all times during the execution. In order to state this property simply, we first define some terms. We say that two or more closures are *siblings* if they were spawned by the same parent, or if they are successors (by one or more `spawn_next`'s) of closures spawned by the same parent. Sibling closures can be ordered by age: the first child spawned is older than the second, and so on. At any given time during the execution, we say that a closure is a *leaf* if it has no allocated children, and we say that a leaf closure is a *primary leaf* if, in addition, it has no younger siblings allocated. The *busy-leaves property* states that every primary-leaf closure has a processor working on it.

Lemma 1 *Cilk's scheduler maintains the busy-leaves property.*

Proof: Consider the three possible ways that a primary-leaf closure can be created. First, when a thread spawns children, the youngest of these children is a primary leaf. Second, when a thread completes and its closure is freed, if that closure has an older sibling and that sibling has no children, then the older-sibling closure becomes a primary leaf. Finally, when a thread completes and its closure is freed, if that closure has no allocated siblings, then the youngest closure of its parent's successor threads is a primary leaf. The induction follows by observing that in all three of these cases, Cilk's scheduler guarantees that a processor works on the new primary leaf. In the third case we use the important fact that a newly activated closure is posted on the processor that activated it (and not on the processor on which it was residing). ■

Theorem 2 *For any fully strict Cilk program, if S_1 is the space used to execute the program on 1 processor, then with any number P of processors, Cilk's work-stealing scheduler uses at most S_1P space.*

Proof: We shall obtain the space bound $S_P \leq S_1P$ by assigning every allocated closure to a primary leaf such that the total space of all closures assigned to a given primary leaf is at most S_1. Since Lemma 1 guarantees that all primary leaves are busy, at most P primary-leaf closures can be allocated, and hence the total amount of space is at most S_1P.

The assignment of allocated closures to primary leaves is made 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 youngest child. If the closure is a leaf but has some younger siblings, then the closure is assigned to the same primary leaf as its youngest 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 are now ready to analyze execution time. Our strategy is to mimic the theorems of [4] for a more restricted model of multithreaded computation. As in [4], the bounds 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 [33]. For technical reasons in our analysis of execution time, the critical path is calculated assuming that all threads spawned by a parent thread are spawned at the end of the parent thread.

In our analysis of execution time, we use an accounting argument. At each time step, each of the P processors places a dollar in one of three buckets according to its actions at that step. If the processor executes an instruction of a thread at the step, it places its dollar into the WORK bucket. If the processor initiates a steal attempt, it places its dollar into the STEAL bucket. Finally, if the processor merely waits for a steal request that is delayed by contention, then it places its dollar into the WAIT bucket. We shall derive the running time bound by upper bounding the dollars in each bucket at the end of the computation, summing these values, and then dividing by P, the total number of dollars put into buckets on each step.

Lemma 3 *When the execution of a fully strict Cilk computation with work T_1 ends, the* WORK *bucket contains T_1 dollars.*

Proof: The computation contains a total of T_1 instructions. ■

Lemma 4 *When the execution of a fully strict Cilk computation ends, the expected number of dollars in the* WAIT *bucket is less than the number of dollars in the* STEAL *bucket.*

Proof: Lemma 5 of [4] shows that if P processors make M random steal requests during the course of a computation, where requests with the same destination are serially queued at the destination, then the expected total delay is less than M. ■

Lemma 5 *When the P-processor execution of a fully strict Cilk computation with critical-path length T_∞ and for which each thread has at most one successor ends, the expected number of dollars in the* STEAL *bucket is $O(PT_\infty)$.*

Proof sketch: The proof follows the delay-sequence argument of [4], but with some differences that we shall point out. Full details can be found in [3], which generalizes to the situation in which a thread can have more than one successor.

At any given time during the execution, we say that a thread is *critical* if it has not yet been executed but all of its predecessors in the dag have been executed. For this argument, the dag must be augmented with "ghost" threads and additional edges to represent implicit dependencies imposed by the Cilk scheduler. We define a *delay sequence* to be a pair (P, s) such that P is a path of threads in the augmented dag and s is a positive integer. We say that a delay sequence (P, s) *occurs* in an execution if at least s steal attempts are initiated while some thread of P is critical.

The next step of the proof is to show that if at least s steal attempts occur during an execution, where s is sufficiently large, then some delay sequence (P, s) must occur. That is, there must be some path P in the dag such that each of the s steal attempts occurs while some thread of P is critical. We do not give the construction here, but rather refer the reader to [3, 4] for directly analogous arguments.

The last step of the proof is to show that a delay sequence with $s = \Omega(PT_\infty)$ is unlikely to occur. The key to this step is a lemma, which describes the structure of threads the processors' ready pools. This structural lemma implies that if a thread is critical, it is the next thread to be stolen from the pool in which it resides. Intuitively, after P steal attempts, we expect one of these attempts to have targeted the processor in which the critical thread of interest resides. In this case, the critical thread will be stolen and executed, unless, of course, it has already been executed by the local processor. Thus, after PT_∞ steal attempts, we expect all threads on P to have been executed. The delay-sequence argument formalizes this intuition. Thus, the expected number s of dollars in the STEAL bucket is at most $O(PT_\infty)$. ∎

Theorem 6 *Consider any fully strict Cilk computation with work T_1 and critical-path length T_∞ such that every thread spawns at most one successor. With any number P of processors, Cilk's work-stealing scheduler runs the computation in expected time $O(T_1/P + T_\infty)$.*

Proof: We sum the dollars in the three buckets and divide by P. By Lemma 3, the WORK bucket contains T_1 dollars. By Lemma 4, the WAIT bucket contains at most a constant times the number of dollars in the STEAL bucket, and Lemma 5 implies that the total number of dollars in both buckets is $O(PT_\infty)$. Thus, the sum of the dollars is $T_1 + O(PT_\infty)$, and the bound on execution time is obtained by dividing by P. ∎

In fact, it can be shown using the techniques of [4] that for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time on P processors is $O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$.

Theorem 7 *Consider any fully strict Cilk computation with work T_1 and critical-path length T_∞ such that every thread spawns at most one successor. For any number P of processors, the total number of bytes communicated by Cilk's work-stealing scheduler has expectation $O(PT_\infty S_{\text{max}})$, where S_{max} is the size in bytes of the largest closure in the computation.*

Proof: The proof follows directly from Lemma 5. All communication costs can be associated with steals or steal requests, and at most $O(S_{\text{max}})$ bytes are communicated for each successful steal. ∎

In fact, for any $\epsilon > 0$, the probability is at least $1 - \epsilon$ that the total communication incurred is $O(P(T_\infty + \lg(1/\epsilon))S_{\text{max}})$.

The analysis and bounds we have derived apply to fully strict programs in the case when each thread spawns at most one successor. Some programs, such as $\star$Socrates, contain threads that spawn several successors. In [3], the theorems above are generalized to handle this situation as follows. Let n_l denote the maximum number of threads belonging to any one procedure such that all the threads are simultaneously living during some execution. Let n_d denote the maximum number of dependency edges between any pair of threads. When each thread can spawn at most one successor, we have $n_l = 1$ and $n_d = 1$ and the theorems as proved in this paper hold. When n_l or n_d exceeds 1, however, the arguments must be modified.

Specifically, when n_l or n_d exceeds 1, the analysis of the number of dollars in the STEAL bucket must be modified. A critical thread may no longer be the first thread to be stolen from a processor's ready pool. Other noncritical threads from the same procedure may be stolen in advance of the critical thread. Moreover, extra dependency edges may cause even more steal attempts to occur before a critical thread gets stolen. Accounting for these extra steals in the argument, we obtain the following bounds on time and communication. For any number P of processors, the expected execution time is $O(T_1/P + n_l T_\infty)$, and the expected number of bytes communicated is $O(n_l P T_\infty (n_d + S_{\max}))$. The $O(S_1 P)$ bound on space is unchanged. Analogous high-probability bounds for time and communication can be found in [3].

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 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 observable quantities, and it delivers guaranteed performance as a function of these quantities. Work and critical-path length have been used in the theory community for years to analyze parallel algorithms [28]. 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-thread-per-processor, 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 Halbherr, now of the Boston Consulting Group in Zurich, Switzerland. Mike's PCM runtime system [20] 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. Xinmin Tian of McGill University provided helpful suggestions for improving the paper. Don Dailey and International Master Larry Kaufman, both formerly of Heuristic Software, were part of the ⋆Socrates development team. 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 SMP platforms. 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 challenge us.

References

[1] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 95–109, Pacific Grove, California, Oct. 1991.

[2] Blelloch, G. E. Programming parallel algorithms. In *Proceedings of the 1992 Dartmouth Institute for Advanced Graduate Studies (DAGS) Symposium on Parallel Computation*, pp. 11–18, Hanover, New Hampshire, Jun. 1992.

[3] Blumofe, R. D. *Executing Multithreaded Programs Efficiently*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sep. 1995.

[4] Blumofe, R. D. and Leiserson, C. E. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 356–368, Santa Fe, New Mexico, Nov. 1994.

[5] Blumofe, R. D. and Park, D. S. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, pp. 96–105, San Francisco, California, Aug. 1994.

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

[7] Brewer, E. A. and Blumofe, R. Strata: A multi-layer communications library. Technical Report to appear, MIT Laboratory for Computer Science. Available as `ftp://ftp.lcs.mit.edu/pub/supertech/strata/strata.tar.Z`.

[8] Burton, F. W. and Sleep, M. R. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pp. 187–194, Portsmouth, New Hampshire, Oct. 1981.

[9] Carlisle, M. C., Rogers, A., Reppy, J. H., and Hendren, L. J. Early experiences with Olden. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, Aug. 1993.

[10] Chandra, R., Gupta, A., and Hennessy, J. L. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, Aug. 1994.

[11] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 147–158, Litchfield Park, Arizona, Dec. 1989.

[12] Cooper, E. C. and Draves, R. P. C Threads. Tech. Rep. CMU-CS-88-154, School of Computer Science, Carnegie-Mellon University, Jun. 1988.

[13] Culler, D. E., Sah, A., Schauser, K. E., von Eicken, T., and Wawrzynek, J. 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*, pp. 164–175, Santa Clara, California, Apr. 1991.

[14] Feldmann, R., Mysliwietz, P., and Monien, B. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 94–103, Cape May, New Jersey, Jun. 1994.

[15] Finkel, R. and Manber, U. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, Apr. 1987.

[16] Freeh, V. W., Lowenthal, D. K., and Andrews, G. R. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 201–213, Monterey, California, Nov. 1994.

[17] Goldstein, S. C., Schauser, K. E., and Culler, D. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.

[18] Graham, R. L. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

[19] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, Mar. 1969.

[20] Halbherr, M., Zhou, Y., and Joerg, C. F. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, Sep. 1994.

[21] Halstead, Jr., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[22] Hillis, W. and Steele, G. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[23] Hsieh, W. C., Wang, P., and Weihl, W. E. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 239–248, San Diego, California, May 1993.

[24] Jagannathan, S. and Philbin, J. A customizable substrate for concurrent languages. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 55–67, San Francisco, California, Jun. 1992.

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

[26] Kalé, L. V. The Chare kernel parallel programming system. In *Proceedings of the 1990 International Conference on Parallel Processing, Volume II: Software*, pp. 17–25, Aug. 1990.

[27] Karamcheti, V. and Chien, A. Concert—efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing '93*, pp. 598–607, Portland, Oregon, Nov. 1993.

[28] Karp, R. M. and Ramachandran, V. Parallel algorithms for shared-memory machines. In van Leeuwen, J., (Ed.), *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity*, chapter 17, pp. 869–941. MIT Press, Cambridge, Massachusetts, 1990.

[29] Karp, R. M. and Zhang, Y. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, Jul. 1993.

[30] Kranz, D. A., Halstead, Jr., R. H., and Mohr, E. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 81–90, Portland, Oregon, Jun. 1989.

[31] Kuszmaul, B. C. *Synchronized MIMD Computing*. Ph.D. 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`.

[32] Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., Feynman, C. R., Ganmukhi, M. N., Hill, J. V., Hillis, W. D., Kuszmaul, B. C., Pierre, M. A. S., Wells, D. S., Wong, M. C., Yang, S.-W., and Zak, R. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 272–285, San Diego, California, Jun. 1992.

[33] Liu, P., Aiello, W., and Bhatt, S. An atomic model for message-passing. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 154–163, Velen, Germany, Jun. 1993.

[34] Miller, R. C. 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.

[35] Mohr, E., Kranz, D. A., and Halstead, Jr., R. H. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul. 1991.

[36] Nikhil, R. S. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pp. 390–405, Portland, Oregon, Aug. 1993. Springer-Verlag.

[37] Nikhil, R. S. Cid: A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Aug. 1994.

[38] Pande, V. S., Joerg, C. F., Grosberg, A. Y., and Tanaka, T. Enumerations of the hamiltonian walks on a cubic sublattice. *Journal of Physics A*, 27, 1994.

[39] Rinard, M. C., Scales, D. J., and Lam, M. S. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, Jun. 1993.

[40] Rudolph, L., Slivkin-Allalouf, M., and Upfal, E. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 237–245, Hilton Head, South Carolina, Jul. 1991.

[41] Sunderam, V. S. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[42] Tanenbaum, A. S., Bal, H. E., and Kaashoek, M. F. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pp. 5–12, Spokane, Washington, Jul. 1993.

[43] Vandevoorde, M. T. and Roberts, E. S. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, Aug. 1988.

[44] Wu, I.-C. and Kung, H. T. Communication complexity for parallel divide-and-conquer. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pp. 151–162, San Juan, Puerto Rico, Oct. 1991.

Short biographies of the authors

ROBERT (BOBBY) BLUMOFE received his Bachelor's degree from Brown University in 1988 and his Ph.D. from MIT in 1995. He started his research career working on computer graphics with Andy van Dam at Brown, and did his Ph.D. work on algorithms and systems for parallel multithreaded computing with Charles Leiserson at MIT. As part of this dissertation work, Bobby developed an adaptive and fault tolerant version of Cilk, called Cilk-NOW, that runs on networks of workstations. Bobby is now an Assistant Professor at the University of Texas at Austin, and he is continuing his work on Cilk and Cilk-NOW.

CHRISTOPHER F. JOERG received the B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from MIT in 1987, 1990, and 1996 respectively. His earlier work at MIT includes the architecture, design, and implementation of the PaRC packet switched routing chip, and the design and analysis of tightly coupled processor-network interface architectures. He was also the lead programmer on the ⋆Socrates massively parallel chess system. Chris is currently on the research staff at Digital's Cambridge Research Lab. His research interests are in the areas of computer architecture, parallel systems, multithreading, and interconnection networks.

BRADLEY C. KUSZMAUL received two S.B. degrees in 1984, an S.M. degree in 1986, and a Ph.D. degree in 1994, all from MIT. In 1987, midway through his Ph.D. program, he took a year off from MIT to serve as one of the principal architects of the Connection Machine CM5 at Thinking Machines Corporation. When he returned to MIT to finish graduate school, he authored the StarTech massively parallel chess program and then coauthored ⋆Socrates. In 1995, he joined the Departments of Computer Science and Electrical Engineering at Yale University, where he is now Assistant Professor. Prof. Kuszmaul's work in solving systems problems in high-performance computing spans a wide range of technology including VLSI chips, interconnection networks, operating systems, compilers, interpreters, algorithms, and applications.

CHARLES E. LEISERSON is Professor of Computer Science and Engineering in the MIT Laboratory for Computer Science. He received the B.S. degree in computer science and mathematics from Yale University in 1975 and the Ph.D. degree in computer science from Carnegie Mellon University in 1981. Prof. Leiserson's research contributions include systolic computing, VLSI graph layout, the retiming method of digital circuit optimization, and the fat-tree interconnection network. He was principal architect of the networks of Thinking Machines' Connection Machine

Model CM5 Supercomputer. He has designed and engineered many parallel algorithms, including ones for matrix linear algebra, graph algorithms, optimization, and sorting. He and his students have won prizes in international competition for their parallel chess programs StarTech and ⋆Socrates. He is coauthor of the textbook *Introduction to Algorithms*, which is published jointly by The MIT Press and McGraw-Hill. Prof. Leiserson is a member of IEEE, ACM, and SIAM, and serves as General Chair for the ACM Symposium on Parallel Algorithms and Architectures.

KEITH H. RANDALL received B.S. degrees in Mathematics and Computer Science from MIT in 1993, and an M.S. degree in Computer Science from MIT in 1994. He expects to receive his Ph.D. from MIT in 1997. His research interests include routing, parallel algorithms, and scheduling.

YULI ZHOU received his B.S. in Electric Engineering from the University of Science and Technology of China in 1983, and M.S. and Ph.D. in Computer Science from the Graduate School of Arts and Sciences, Harvard University in 1990. Since then he has worked at the MIT Laboratory for Computer Science as a research associate on compilers for parallel programming languages in the Computation Structures Group. He is currently a member of the technical staff at AT&T Bell Laboratories. His main research interests are programming languages and parallel/distributed computing.