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, backtrack 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 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 is constrained to follow the precedence relation determined by the dag of threads.

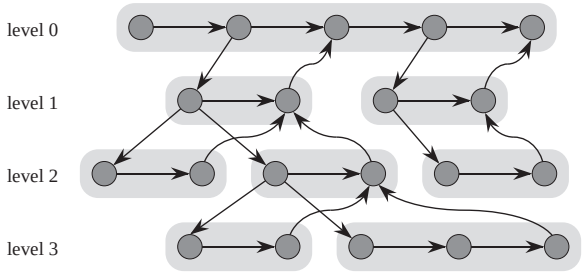

Figure 1: The Cilk model of multithreaded computation. Threads are shown as circles, which are grouped into procedures. Each downward edge corresponds to a spawn of a child, each horizontal edge corresponds to a spawn of a successor, and each upward, curved edge corresponds to a data dependency. The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

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_∞, 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

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 is supported in part by an ARPA High-Performance Computing Graduate Fellowship. Keith Randall is supported in part by a Department of Defense NDSEG Fellowship.

To appear in the Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara California, July 19–21, 1995. Also available on the web as `ftp://ftp.lcs.mit.edu/pub/cilk/PPoPP95.ps.Z`.

execution time cannot be less than T_1/P or less than T_∞. 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 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

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:

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

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 `cont`. A continuation

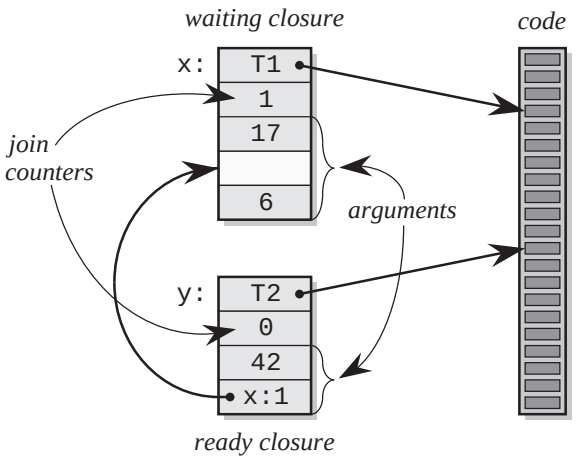

Figure 2: The closure data structure.

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:

```
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 as in C. 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 usually created with no missing arguments.

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 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

```
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.

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_next`'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_next` 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 communication 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 benchmark 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 interconnection 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 7 levels of the search tree.
- `pfold` is a protein-folding program [35] written in conjunction 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 program 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.
- $\star$Socrates is a parallel chess program that uses the Jamboree search algorithm [23, 29] 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 is a production-quality program that won third prize in the 1994 ACM International Computer Chess Championship running on the 512-node CM5 in the National Center for Supercomputing Applications at the University of Illinois, Urbana-Champaign.

Table 4 shows typical performance measures for these 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 work T_1 is the measured execution time for the Cilk program running on a single node of the CM5.[2] The critical path length T_∞ of the Cilk computation is measured by timestamping each thread and does not include scheduling or communication costs. The measured P-processor execution time of the Cilk program running on the CM5 is given by T_P, which includes all scheduling and communication costs. 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 in the next section. 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 Table 4 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 scheduler induces very 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 [29] 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,

[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 $\star$Socrates program, T_1 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. $\star$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.

	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)
				(application 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

Table 4: Performance of Cilk on various applications. All times are in seconds, except where noted.

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.[3] The ⋆Socrates program exhibits somewhat less parallelism and also somewhat less speedup. On 32 processors the ⋆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 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 ⋆Socrates with low efficiency generate correspondingly low application speedup. The ⋆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 performance prediction, we prefer to decouple 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 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 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 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 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 closures. On 256 processors, the number of executed threads nearly doubles to over 51 million, but the space per processors 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,

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

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 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_∞ 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 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_∞. 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, on 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 $c_1 = 0.9543 \pm 0.1775$ 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 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_∞ 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_∞ 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

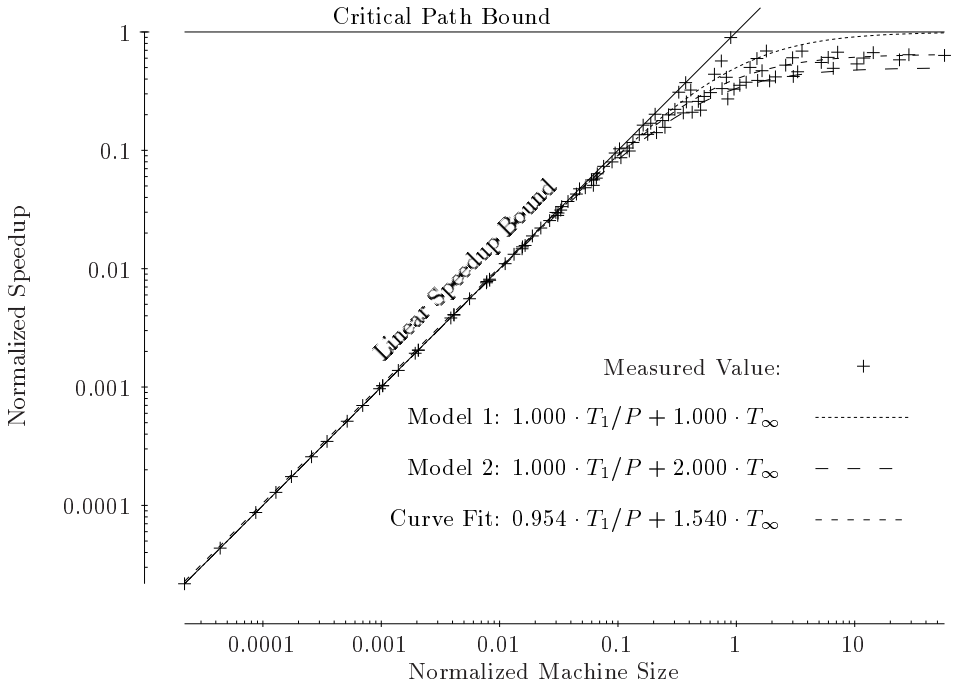

Figure 5: Normalized speedups for the `knary` synthetic benchmark using from 1 to 256 processors. The horizontal axis is P and the vertical axis is the speedup T_1/T_P, but each data point has been normalized by dividing the these parameters by T_1/T_∞.

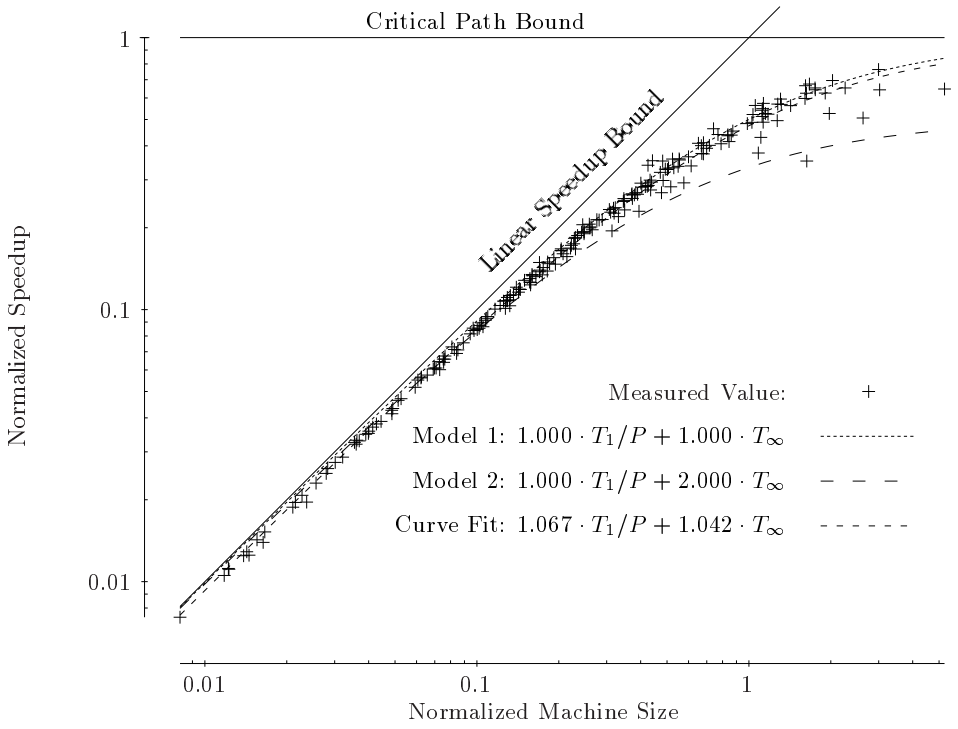

Figure 6: Normalized speedups for the ⋆Socrates chess program.

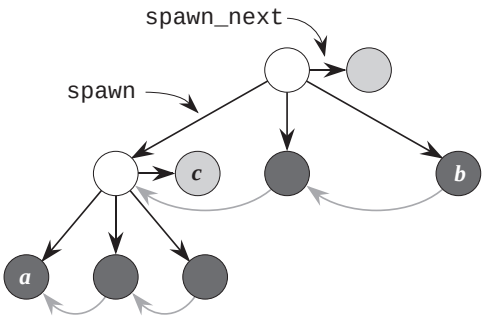

Figure 7: The closures at some time during a 1-processor execution. Data-dependency edges are not shown. The black nodes represent ready closures, the gray nodes represent waiting closures, and white nodes represent closures that have already been executed. The black and gray closures are allocated and consume space, but the white closures have been deallocated. Gray, curved edges represent the additional edges in D' that do not also belong to D.

additive term in both cases. Proofs of these bounds use generalizations of the techniques developed in [3]. We defer complete proofs and give outlines here.

The space bound follows from the "busy-leaves" property which characterizes the allocated closures at all times during the execution. At any given time during the execution, we say that a closure is a *leaf* if it has no allocated child closures, and we say that a leaf closure is a *primary leaf* if, in addition, it has no left-sibling closures allocated. In Figure 7, which shows the allocated closures at some time during an execution, closure a is the only primary leaf. Closure b is a leaf, but it is not primary, since it has left siblings and closure c is not a leaf, because a and its two siblings are counted as children of c. The *busy-leaves* property states that every primary leaf closure has a processor working on it. To prove the space bound, we show that Cilk's scheduler maintains the busy-leaves property, and then we show that the busy-leaves property implies the space bound.

Theorem 1 *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 $S_1 P$ space.*

Proof: We first show by induction on execution time that Cilk's work-stealing scheduler maintains the busy-leaves property. We then show that the busy-leaves property implies the space bound.

To see that Cilk's scheduler maintains the busy-leaves property, we consider the three possible ways that a primary-leaf closure can be created. First, when a thread spawns children, the leftmost of these children is a primary leaf. Second, when a thread completes and its closure is freed, if that closure has a right sibling and that sibling has no children, then the right-sibling closure becomes a primary leaf. And third, when a thread completes and its closure is freed, if that closure has no allocated siblings, then the leftmost 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 fact that a newly activated closure is posted on the processor that activated it and not on the processor on which it was residing.

The space bound $S_P \leq S_1 P$ is obtained by showing that every allocated closure can be associated with a primary leaf and that the total space of all closures assigned to a given primary leaf is at most S_1. Since Cilk's scheduler keeps all primary leaves busy, with P processors we are guaranteed that at every time during the execution, at most P primary-leaf closures can be allocated, and hence the total amount of space is at most $S_1 P$.

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_∞, then Cilk's work-stealing scheduler executes the program in expected time $\mathrm{E}[T_P] = O(T_1/P + T_\infty)$. Furthermore, for any $\epsilon > 0$, the execution time is $T_P = O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$ 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_∞ and maximum closure size $S_{\max}$, then Cilk's work-stealing scheduler incurs expected communication $O(T_\infty P S_{\max})$. Furthermore, for any $\epsilon > 0$, the communication cost is $O((T_\infty + \lg(1/\epsilon))PS_{\max})$ 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 $O(T_\infty P)$ steal attempts occur in an execution, and all communication costs can be associated with one of these steal requests such that at most $O(S_{\max})$ 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 Halbherr, 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. Xinmin 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

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

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

[3] 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.

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

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

[6] Eric A. Brewer and Robert Blumofe. 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`.

[7] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages*

and Computer Architecture, pages 187–194, Portsmouth, New Hampshire, October 1981.

[8] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early experiences with Olden. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[9] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[10] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, Arizona, December 1989.

[11] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie-Mellon University, June 1988.

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

[13] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, Cape May, New Jersey, June 1994.

[14] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[15] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

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

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

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

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

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

[21] Wilson C. Hsieh, Paul Wang, and William E. Weihl. 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)*, pages 239–248, San Diego, California, May 1993.

[22] Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 55–67, San Francisco, California, June 1992.

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

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

[25] Vijay Karamcheti and Andrew Chien. Concert—efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing '93*, pages 598–607, Portland, Oregon, November 1993.

[26] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity*, chapter 17, pages 869–941. MIT Press, Cambridge, Massachusetts, 1990.

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

[28] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, Oregon, June 1989.

[29] Bradley C. Kuszmaul. *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`.

[30] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, California, June 1992.

[31] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An atomic model for message-passing. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 154–163, Velen, Germany, June 1993.

[32] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[33] Rishiyur S. Nikhil. 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, pages 390–405, Portland, Oregon, August 1993. Springer-Verlag.

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

[35] Vijay S. Pande, Christopher F. Joerg, Alexander Yu Grosberg, and Toyoichi Tanaka. Enumerations of the hamiltonian walks on a cubic sublattice. *Journal of Physics A*, 27, 1994.

[36] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[37] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, South Carolina, July 1991.

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

[39] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, Spokane, Washington, July 1993.

[40] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

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