Programming Parallel Applications in Cilk

Charles E. Leiserson and Aske Plaat
MIT Laboratory for Computer Science, Cambridge, MA 02139, USA

December 7, 1997

Abstract

Cilk (pronounced "silk") is a C-based language for multithreaded parallel programming. Cilk makes it easy to program irregular parallel applications, especially as compared with data-parallel or message-passing programming systems. A Cilk programmer need not worry about protocols and load balancing, which are handled by Cilk's provably efficient runtime system. Many regular and irregular Cilk applications run nearly as fast on one processor as comparable C programs, and they scale well to many processors.

1 Introduction

Cilk is an algorithmic multithreaded language. The philosophy behind Cilk is that a programmer should concentrate on structuring his program to expose parallelism and exploit locality, leaving the runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. Cilk's runtime system takes care of details like load balancing and communication protocols. Unlike other multithreaded languages, however, Cilk is algorithmic in that the runtime system's scheduler guarantees provably efficient and predictable performance.

The basic Cilk language is extremely simple. It consists of C with the addition of three new keywords to indicate parallelism and synchronization. A Cilk program, when run on one processor, has the same semantics as the C program that results when the Cilk keywords are deleted. In addition, the Cilk system extends serial C semantics in a natural way for parallel execution. For example, C's stack memory is implemented as a "cactus" stack in Cilk.

One of the simplest examples of a Cilk program is a recursive program to compute the nth Fibonacci number. A C program to compute the nth Fibonacci number is shown in Figure 1(a), and Figure 1(b) shows a Cilk program that does the same computation in parallel.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-94-1-0985. Aske Plaat was supported in part by a postdoctoral fellowship from the Erasmus University Rotterdam, the Netherlands. An early version of this paper appeared in the *Fourth International Symposium on Solving Irregularly Structured Problems In Parallel* (*IRREGULAR'97*), June 1997, University of Paderborn, Germany.

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


int fib (int n)
{
    if (n<2) return (n);
    else
    {
        int x, y;
        x = fib (n-1);
        y = fib (n-2);

        return (x+y);
    }
}

int main (int argc, char *argv[])
{
    int n, result;
    n = atoi(argv[1]);
    result = fib (n);

    printf ("Result: %d\n", result);
    return 0;
}
```

(a)

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

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

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

(b)

Figure 1: Programs to compute the nth Fibonacci number. **(a)** A serial C program. **(b)** A parallel Cilk program.

Notice how similar the two programs look. In fact, the only differences between them are the inclusion of the library header file and the Cilk keywords `cilk`, `spawn`, and `sync`.

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

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

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

Cilk uses a *cactus stack* [8] for stack-allocated storage, such as is needed for procedure-local variables. As is shown in Figure 2, from the point of view of a single Cilk procedure, a cactus stack behaves much like an ordinary stack. The procedure can allocate and free memory by pushing and popping the stack. The procedure views the stack as extending back from its own stack frame to the frame of its parent and continuing to more distant ancestors. The stack becomes a cactus stack when multiple procedures execute in parallel, each with its own view of the stack that corresponds to its call history, as shown in Figure 2.

Cactus stacks in Cilk have essentially the same limitations as ordinary C stacks [8]. For instance, a child procedure cannot return to its parent a pointer to an object that it has allocated, since the object will be deallocated automatically when the child returns. Similarly, sibling procedures cannot reference each other's local variables. Just as with the C stack, pointers to objects allocated on the cactus stack can only be safely passed to procedures below the allocation point in the call tree. Cilk supports heap memory as well as stack memory, however, and a Cilk `malloc()` function is available to programmers.

The Cilk language also supports several advanced parallel programming features. It provides "inlets" as a means of incorporating a returned result of child procedures into a procedure

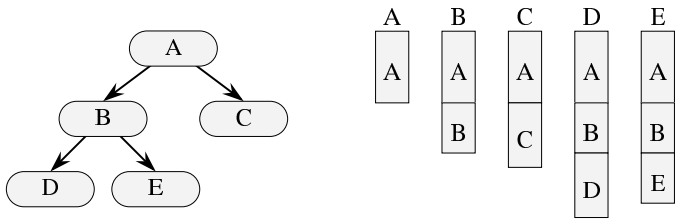

Figure 2: A cactus stack. Procedure A spawns B and C, and B spawns D and E. The left part of the figure shows the spawn tree, and the right part of the figure shows the view of the stack by the five procedures. (The stack grows downward.)

frame in nonstandard ways. Cilk also allows a procedure to abort speculatively spawned work. A procedure can also interact with Cilk's scheduler to test whether it is "synched" without actually executing a `sync`. The Cilk-5 reference manual [9] provides complete documentation of the Cilk language.

2 The Cilk model of multithreaded computation

Cilk supports an algorithmic model of parallel computation. Specifically, it guarantees that programs are scheduled efficiently by its runtime system. To better understand this guarantee, this section surveys the major characteristics of Cilk's algorithmic model.

A Cilk program execution consists of a collection of *procedures*,[1] each of which is broken into a sequence of nonblocking *threads*. In Cilk terminology, a *thread* is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` statement. (The evaluation of arguments to these statements is considered part of the thread preceding the statement.) The first thread that executes when a procedure is called is the procedure's *initial thread*, and the subsequent threads are *successor threads*. At runtime, the binary "spawn" relation causes procedure instances to be structured as a rooted tree, and the dependencies among their threads form a dag embedded in this *spawn tree*, as is illustrated in Figure 3.

A correct execution of a Cilk program must obey all the dependencies in the dag, since a thread cannot be executed until all the threads on which it depends have completed. These dependencies form a partial order, permitting many ways of scheduling the threads in the dag. The order in which the dag unfolds and the mapping of threads onto processors are crucial decisions made by Cilk's scheduler. Every active procedure has associated state that requires storage, and every dependency between threads assigned to different processors requires communication. Thus, different scheduling policies may yield different space and time requirements for the computation.

It can be shown that for general multithreaded dags, no good scheduling policy exists. That is, a dag can be constructed for which any schedule that provides linear speedup also requires vastly more than linear expansion of space [4]. Fortunately, every Cilk program

[1] Technically, procedure *instances*.

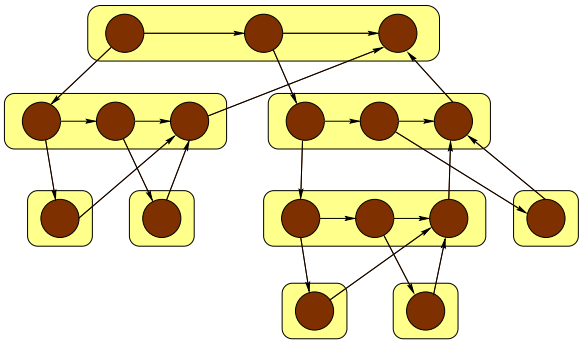

Figure 3: The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates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

generates a well-structured dag that can be scheduled efficiently [5].

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

Cilk's work-stealing scheduler executes any Cilk computation in nearly optimal time. From an abstract theoretical perspective, there are two fundamental limits as to how fast a Cilk program can run. Let us denote by T_P the execution time of a given computation on P processors. The *work* of the computation is the total time needed to execute all threads in the dag. We can denote the work by T_1, since the work is essentially the execution time of the computation on one processor. Notice that with T_1 work and P processors, the lower bound $T_P \geq T_1/P$ must hold.[2] The second limit is based on the program's *critical-path length*, denoted by T_∞, which is the execution time of the computation on an infinite number of processors, or equivalently, the time needed to execute threads along the longest path of dependency. The second lower bound is simply $T_P \geq T_\infty$.

Cilk's work-stealing scheduler executes a Cilk computation on P processors in time $T_P \leq T_1/P + O(T_\infty)$, which is asymptotically optimal. Empirically, the constant factor hidden by the big O is often close to 1 or 2 [3], and the formula

$$T_P \approx T_1/P + T_\infty \tag{1}$$

[2]This abstract model of execution time ignores memory-hierarchy effects, but is nonetheless quite accurate [3].

is a good approximation of runtime. (This model assumes that the parallel computer has adequate bandwidth in its communication network.) This performance model can be interpreted using the notion of *average parallelism*, which is given by the formula $\overline{P} = T_1/T_\infty$. The average parallelism is the average amount of work for every step along the critical path. Whenever $P \ll \overline{P}$, meaning that the actual number of processors is much smaller than the average parallelism of the application, we have equivalently that $T_1/P \gg T_\infty$. Thus, the model predicts that $T_P \approx T_1/P$ and the Cilk program is guaranteed to run with almost perfect linear speedup. The measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs over the entire range of possible parallel machine sizes. Cilk provides automatic timing instrumentation that can calculate these two measures during a run of a program, no matter how many are used to run the program.

Cilk's runtime system also provides a guarantee on the amount of cactus stack space used by a parallel Cilk execution. Denote by S_P the (cactus) stack space required for a P-processor execution. Then, S_1 is the space required for an execution on one processor. Cilk's scheduler guarantees that for a P-processor execution, we have $S_P \leq S_1 P$, which is to say that the average space per processor is bounded above by the serial space. In fact, much less space may be required for many algorithms (see [2]), but the bound $S_P \leq S_1 P$ serves as a reasonable limit. If a computation uses moderate amounts of memory when on one processor, one can be assured that it will use no more space per processor when run in parallel.

The algorithmic complexity measures of work, critical-path length, and space—together with the fact that a programmer can count on them when designing a program—justify Cilk as an *algorithmic* multithreaded language.

3 Experiments

The Cilk distribution contains a variety of example programs which explore the difficulty of solving problems in parallel. Some of these programs, such as the program for computing a sparse Cholesky factorization, have irregular inputs. Others, like the backtrack searching algorithm used to solve the n-queens problem, have an irregular structure in the computation. Because of Cilk's flexibility in expressing parallelism, irregular problems pose no undue hardship on execution efficiency. The minimal loss of performance that is sometimes experienced is generally due to parallel algorithms that are intrinsically less efficient than the serial algorithm they replace. This section describes some preliminary performance measurements taken of the example programs.

The Cilk distribution (available from `http://theory.lcs.mit.edu/~cilk`) includes the following programs:

- `blockedmul` — Blocked multiplication of two dense $n \times n$ matrices, written by Keith Randall.
- `notempmul` — A slightly less parallel, but more efficient, blocked multiplication of two dense $n \times n$ matrices, written by Keith Randall.
- `strassen` — Strassen's algorithm for multiplication of two dense $n \times n$ matrices, written

Program	*Size*	T_1	T_∞	$\overline{P}$	T_1/T_S	T_8	T_1/T_8	T_S/T_8
`blockedmul`	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
`notempmul`	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
`strassen`	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
`cilksort`*	4,100,000	5.4	0.0049	1108	1.21	0.90	6.0	5.0
`queens`†	22	150	0.0015	96898	0.99	18.8	8.0	8.0
`knapsack`†	30	682	0.0017	392343	1.21	85	8.0	6.6
`lu`*	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
`cholesky`*	BCSSTK29	87	0.64	136	1.22	18	4.8	3.9
—	BCSSTK32	1427	3.4	420	1.25	208	6.9	5.5
`heat`	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
`fft`*	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
`Barnes-Hut`	2^{16}	124	8.3	15	1.02	25	5.0	4.9

Table 1: The performance of example Cilk programs. Times are in seconds. Measurements are for a complete run of the program, except for those programs that are starred (*), where because of large setup times, only the core algorithm was measured. Programs labeled by a dagger (†) are nondeterministic, and thus, the running time on one processor is not the same as the work performed by the computation. For these programs, the value for T_1 indicates the actual work of the computation, and not the running time on one processor.

by Michael Bender, Stuart Schechter, and Bin Song.

- `cilksort` — Sort a random permutation of n 32-bit integers, written by Matteo Frigo and Andrew Stark.
- `queens` — Backtrack search to solve the problem of placing n queens on an $n \times n$ chessboard so that no two queens attack each other, written by Keith Randall.
- `knapsack` — Solve the 0-1 knapsack problem on n items using branch and bound, written by Matteo Frigo.
- `lu` — LU-decomposition (without pivoting) of a dense $n \times n$ matrix, written by Robert D. Blumofe.
- `cholesky` — Cholesky factorization of a sparse symmetric positive-definite matrix represented as a quad-tree, written by Aske Plaat and Keith Randall.
- `heat` — Heat-diffusion calculation on an $m \times n$ mesh, written by Volker Strumpen.
- `fft` — Fast Fourier transformation of a vector of length n, written by Matteo Frigo.
- `Barnes-Hut` — Barnes-Hut n-body calculation, written by Keith Randall.

Figure 1 shows speedup measurements that were taken of the programs, as well as measurements of work T_1, critical path T_∞, and average parallelism $\overline{P} = T_1/T_\infty$. The machine used for the test runs was an otherwise idle Sun Enterprise 5000 SMP, with 8 167-megahertz UltraSPARC processors, 512 megabytes of main memory, 512 kilobytes of L2 cache, 16 kilobytes of instruction L1 cache, and 16 kilobytes of data L1 cache, running Solaris 2.5 and a version of Cilk-5 that used `gcc` 2.7.2 with optimization level `-O3`. The times measured are those of a complete run, except for `cilksort`, `lu`, `cholesky`, and `fft` (which are starred in the figure). For these codes, the setup time to read in the input was sufficiently long compared

to the runtime that only the core algorithm was measured. For `cholesky`, numbers for two sparse matrices from the Harwell-Boeing test set [6] are reported. The matrix BCSSTK29 has dimension 13992, with 619488 nonzeros, or 0.3 percent of the matrix entries. The BCSSTK32 matrix has dimension 44609, with 1029655 nonzeros, or 0.05 percent of the entries. The two matrices were ordered using Matlab's minimum-degree ordering heuristic, which is not included in the `cholesky` benchmark. Regrettably, our time measurements are only accurate to within about 10 percent, due to unpredictability in today's deeply pipelined processor architectures, caused, for example, by direct-mapped caches and instruction alignment.

The column T_1/T_S gives the overhead of the 1-processor Cilk run versus our best serial C algorithm, showing that the overhead imposed by the Cilk runtime system is generally small. The T_8 column gives the time in seconds for a 8-processor run. The speedup column T_1/T_8 gives the time of the 8-processor run of the parallel program compared to that of the 1-processor run (or work, in the case of the nondeterministic programs) of the same parallel program. (The measurements for `queens` and `fft`, which show a speedup for the Cilk implementation over the C implementation, are likely caused by a difference in code alignment in the instruction prefetch buffer.) The T_S/T_8 column gives the speedup relative to the C code.

Two of the example programs, `queens` and `knapsack`, which are marked by a dagger (†) in the figure, are nondeterministic programs. The work of these programs depends on how they are scheduled. For these programs, the figures in the column labeled T_1 (and the other dependent figures) give the work in the computation as measured by adding up the individual execution times of each of the threads, rather than as measured by a one-processor run, as would otherwise be implied. For the other (deterministic) programs, the measures of T_1 and work are synonymous. Because Cilk reports work and critical path measurements, it enables meaningful speedup measurements of programs whose work depends on the actual runtime schedule. Conventionally, speedup is calculated as the one-processor execution time divided by the parallel execution time. This methodology, while correct for deterministic programs, can lead to misleading results for nondeterministic programs, since two runs of the same program can actually be different computations. Cilk's instrumentation can compute the work on any number of processors by adding up the execution times of individual threads, thereby allowing speedup to be calculated properly for nondeterministic programs.

As can be seen from the table, all of the programs exhibit generally good speedups. Even the complicated and irregular `Barnes-Hut` code achieves a speedup of 4.9 on 8 processors, which is at least as good as any implementation we have found in the literature or on the World Wide Web. Furthermore, as can be seen in the T_1/T_S column, the performance of any of our Cilk programs running on one processor is generally indistinguishable from the performance of the comparable C code. The sorting example and sparse Cholesky factorization are worst cases for this set of examples, and even for these programs, the single-processor Cilk performance is within 25 percent of our fastest C code for the problem. The slowdown for sorting is due to the fact that unlike a good serial quicksort, our parallel algorithm cannot be performed in place. The slowdown of the Cholesky factorization is due the overhead in our quad-tree representation of sparse matrices.

As a final note on the performance of Cilk, we mention that there is one unfortunate

aspect of Cilk's current dependence on the otherwise outstanding `gcc` compiler technology. Some machines have a native C compiler which is heavily optimized to exploit the machine's floating-point capability. We have found that these native compilers can sometimes produce floating-point code that is nearly twice as fast as that produced by `gcc` (although `gcc` remains competitive for integer-dominated calculations). Since our `cilk2c` compiler does not produce ANSI-standard C output, but rather exploits some of the advanced capabilities of `gcc`, we cannot directly take advantage of these native compilers. Consequently, in order to obtain the best performance on programs with heavy use of floating-point, a user must use the native compiler to separately compile C functions containing the floating-point inner loops and link them in with the `gcc`-compiled `cilk2c` output of the rest of his program. We hope to alleviate this inconvenience by eventually providing a new Cilk compiler that translates Cilk into ANSI-standard C.

4 Conclusion

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

Cilk's fork/join parallelism is well suited for expressing divide-and-conquer algorithms. Some algorithms, such as FFT and Cholesky factorization, have traditionally been implemented using for-loops. For efficient implementations, the steep memory hierarchy of a modern computer forces these algorithms to be reformulated in a blocked fashion, making the algorithms harder to understand and reducing their scalability and portability. Divide-and-conquer solutions, which parallelize naturally in Cilk, better exploit the memory hierarchy of today's microprocessors than do for-loops. The natural blocking due to the divide-and-conquer paradigm often allows recursive programs to exploit multilevel caching near optimally without knowing the specific cache sizes.

As a case in point, consider Strassen's algorithm for matrix multiplication, which is a divide-and-conquer algorithm. Conventional wisdom has it that although it is theoretically superior (runs in $\Theta(n^{2.81})$ time) to the ordinary algorithm that employs a triply nested for-loop, the constant factor overheads are so large that in practice it is only suited for very large matrices. Surprisingly, in our measurements, it is 48 percent faster than a blocked version of the traditional algorithm for matrices of moderate size. As memory hierarchies become taller, divide-and-conquer algorithms such as Strassen's will become more and more appealing, especially since both algorithms appear to be about as complex to implement. Moreover, such an algorithm is easy to code in Cilk, which takes care of all the complexities of scheduling and load balancing.

Because the semantics of Cilk are a simple and natural extension of C semantics, solving regular and irregular problems in Cilk imposes insignificant runtime overhead compared to solving the problems in C. Programming in parallel can be harder, however, because to obtain parallelism, one must sometimes change a serial algorithm in a way that sacrifices efficiency. Our initial experiences in writing Cilk programs for regular and irregular problems, however, lead us to believe that this loss of efficiency is frequently small or negligible. Nevertheless, more experience with Cilk will be required to evaluate its effectiveness across a wide range of applications. The Cilk developers invite you to program your favorite application in Cilk.

Cilk software, documentation, publications, and up-to-date information are available via the World Wide Web at `http://theory.lcs.mit.edu/~cilk`. The two MIT Ph.D. theses [1, 7] contain more detailed descriptions of the foundation and history of early Cilk versions. Cilk development is currently being carried out at MIT under the direction of Charles E. Leiserson, in cooperation with Robert D. Blumofe of the University of Texas at Austin.

Acknowledgments

We consider ourselves fortunate to have worked on Cilk with many talented people. Thanks in particular to Mingdong Feng, Matteo Frigo, Phil Lisiecki, Keith Randall, Bin Song, and Volker Strumpen for their contributions to the Cilk-5 release. Thanks to Bobby Blumofe of University of Texas at Austin for his continuing contributions. Michael Halbherr, Chris Joerg, Bradley Kuszmaul, Rob Miller, and Yuli Zhou all made substantial contributions to earlier releases.

References

[1] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

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

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

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

[5] 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 (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[6] Iain Duff, Roger G. Grimes, and John G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report TR/PA/92/86, CERFACS, October 1992.

[7] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[8] Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. Technical Report memo AI-199, MIT Artificial Intelligence Laboratory, June 1970.

[9] Supercomputing Technology Group, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139. *Cilk-5.0 (Beta 1) Reference Manual*, March 1997. Available on the World Wide Web at URL "`http://theory.lcs.mit.edu/~cilk`".