from *SIAM News*, Volume 31, Number 4

Programming Parallel Applications In Cilk

By Charles E. Leiserson and Aske Plaat

Cilk (pronounced "silk") is a C-based algorithmic, multithreaded language for parallel programming being developed at the MIT Laboratory for Computer Science. Cilk makes it easy to program irregular parallel applications, especially as compared with data-parallel or message-passing programming systems. Many regular and irregular Cilk applications run nearly as fast as comparable C programs on one processor, and they scale well to many processors.

Cilk allows a programmer to concentrate on structuring a program to expose parallelism and exploit locality, leaving the runtime system with the responsibility for scheduling the computation to run efficiently on a given platform. The Cilk programmer need not worry about protocols and load balancing, which are handled by Cilk's provably efficient runtime system. Unlike other multithreaded languages, however, Cilk is algorithmic, in that the runtime system's scheduler guarantees provably efficient and predictable performance.

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

Greg Astfalk, Editor

The basic Cilk language consists of C with the addition of three new keywords—**`cilk`**, **`spawn`**, and **`sync`**—to indicate parallelism and synchronization. Figure 1 shows a Cilk program that computes the *n*th Fibonacci number. (This program uses an inefficient, exponential-time algorithm. Although logarithmic-time methods are known, the program nevertheless provides a good didactic example.) If the keywords **`cilk`**, **`spawn`**, and **`sync`** are deleted from a Cilk program, the result is a syntactically and semantically correct C program, which we call the *C elision* of the Cilk program. Cilk is a *faithful* extension of C in that a Cilk program's C elision provides a legal implementation of the parallel semantics.

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

As in C, most of the work in a Cilk procedure is executed serially; 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 as with a C function call, execution proceeds to the child when a Cilk procedure is spawned. For a C function call, however, the parent suspends until its child has returned, whereas for a Cilk spawn, the parent can continue to execute in parallel with the child. Indeed, the parent can continue to spawn children, producing a high degree of parallelism. Cilk's scheduler takes the responsibility for 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 the children have completed. The **`sync`** statement is a local "barrier," as opposed to the global barriers sometimes used in, for example, message-passing programming. In Cilk, a **`sync`** waits only for the spawned children of the procedure to complete. When all the children have returned, 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 the **`spawn`** and **`sync`** keywords to expose the parallelism in a program. The Cilk runtime system takes the responsibility for scheduling the procedures efficiently.

Cilk's runtime system supports C's semantics for stack-allocated storage. A pointer to a local variable can be passed to a subroutine, but a pointer to a local variable cannot be returned, since local variables are

```
#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;
}
```

Figure 1. *A parallel Cilk program for computing the nth Fibonacci number.*

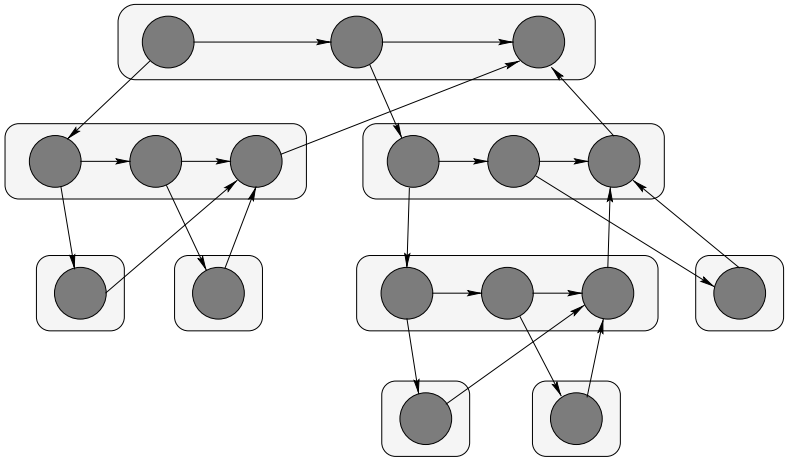

Figure 2. *The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. Edges of all three types are dependencies that constrain the order in which threads can be scheduled.*

deallocated automatically on a return. Cilk supports these features of C's semantics exactly, while allowing subprocedures to execute in parallel. In addition, Cilk supports heap memory through a **`malloc()`** function.

The Cilk language also supports several advanced parallel programming features. "Inlets" give the user nonstandard ways to incorporate a returned result of child procedures into a procedure frame. 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 [8] provides complete documentation of the Cilk language.

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. A Cilk program execution consists of a collection of procedures—technically, procedure instances—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 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; the dependencies among their threads form a directed acyclic graph (dag) embedded in this spawn tree, as illustrated in Figure 2.

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 an 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 an expansion of space that is vastly more than linear [4]. Fortunately, every Cilk program 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, a processor that runs out of work asks another processor, chosen at random, for work to do. Locally, a processor executes procedures in ordinary serial order (just as in 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 (as in 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 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 to the speed at which 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.* 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

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

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

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

is a good approximation of runtime. This model assumes that the parallel computer has adequate bandwidth in its communication network. To interpret this performance model, we can use 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 $P << \overline{P}$, meaning that the actual number of processors is much smaller than the average parallelism of the application, we have equivalently that $T_1/P >> T_\infty$. 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.

Cilk's runtime system also provides a guarantee on the amount of cactus stack space used by a parallel Cilk execution. If the (cactus) stack space required for a P-processor execution is denoted by S_P, 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 run on one processor, we 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 the designation of Cilk as an *algorithmic* multithreaded language.

Experiments

The Cilk distribution (see http://theory.lcs.mit.edu/~cilk) contains a variety of sample programs that explore the difficulty of solving problems in parallel. Some of these programs, such as that for computing a sparse Cholesky factorization, have irregular inputs. Others, like the backtrack searching algorithm used to solve the *n*-queens problem, have irregular structures in the computation. Because of Cilk's flexibility in expressing parallelism, irregular problems pose no undue hardship with respect to execution efficiency. The minimal loss of performance sometimes experienced is generally due to parallel algorithms that are intrinsically less efficient than the serial algorithms they replace. This section describes some preliminary performance measurements for the sample programs.

Table 1 shows speedup measurements for the programs, as well as measurements of work (T_1), critical-path length (T_∞), and average parallelism $\left(\overline{P} = T_1 / T_\infty\right)$. The machine used for the test runs was an otherwise idle Sun Enterprise 5000 SMP, with eight 167-megahertz UltraSPARC processors, 512 megabytes of main memory, 512 kilobytes of L2 cache, 116 kilobytes of instruction L 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 are for complete runs, except for **`cilksort`**, **`lu`**, **`cholesky`**, and **`fft`** (which are starred in the table). For these codes, the time required to read in the input (the setup time) was sufficiently long as compared with 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 13,992, with 619,488, or 0.3%, of the matrix entries being nonzeros. The BCSSTK32 matrix has dimension 44,609, with 1,029,655, or 0.05%, of the entries being nonzeros. Ordering of the two matrices was done with MATLAB's minimum-degree ordering heuristic, which is not included in the **`cholesky`** benchmark. Regrettably, our time measurements are accurate only to within about 10% due to the unpredictability of today's deeply pipelined processor architectures caused, for example, by direct-

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. *Performance of sample Cilk programs. Times are in seconds. Measurements are for a complete run of the program, except for programs labeled with an asterisk (*), for which, because of large setup times, only the core algorithm was measured. Programs labeled with a dagger (†) are nondeterministic, and the runtime on one processor is therefore 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 runtime on one processor.*

mapped caches and instruction alignment.

The column T_1/T_S gives the overhead of the one-processor Cilk run versus that of 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 an eight-processor run. The speedup column, T_1/T_8, gives the time of the eight-processor run of the parallel program compared with that of the one-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 sample programs, **queens** and **knapsack** (marked by a dagger (†) in the table), 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 the sum of the individual execution times of the threads, rather than as the time of a one-processor run, as would otherwise be implied. For the other (deterministic) programs, the measures of T_1 and work are synonymous. By reporting work and critical-path measurements, Cilk makes possible 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 the execution times of individual threads, thereby allowing speedup to be calculated properly for nondeterministic programs.

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

As a final note on the performance of Cilk, we mention one unfortunate aspect of Cilk's current dependence on the otherwise outstanding **gcc** compiler technology. Some machines have native C compilers that are heavily optimized to exploit the machines' 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 code, a user must use the native compiler to separately compile C functions containing the floating-point inner loops and link them with the **gcc**-compiled **cilk2c** output of the rest of the program. We hope to alleviate this inconvenience by eventually providing a new Cilk compiler that translates Cilk into ANSI-standard C.

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 with **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, exploit the memory hierarchy of today's microprocessors better than **for** loops do. The natural blocking that occurs with the divide-and-conquer paradigm often allows recursive programs to exploit multilevel caching almost 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 Strassen's algorithm is theoretically superior (running 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 suited only for very large matrices. Surprisingly, in our measurements, it is 48% faster than a blocked version of the traditional algorithm for matrices of moderate size. As memory hierarchies become taller, divide-and-conquer algorithms like Strassen's will become more and more appealing, especially since the complexity of implementation appears to be about the same for the two algorithms. Moreover, an algorithm like Strassen's 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, solution of regular and irregular problems in Cilk imposes insignificant runtime overhead compared with that incurred in solving the problems in C. Programming in parallel can be harder, however, because obtaining parallelism sometimes involves changing 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.

The Cilk developers are currently working to enhance the Cilk system environment, including support for parallel I/O and streams, job scheduling, and fault tolerance. Cilk software, documentation, publications, and up-to-date information are available via the Web at http://theory.lcs.mit.edu/~cilk. Detailed descriptions of the foundation and history of early Cilk versions can be found in [1, 7].

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 the 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. Research on Cilk has been supported in part by the Defense Advanced Research Projects Agency under grant N00014-94-1-0985. Aske Plaat was supported in part by a postdoctoral fellowship from the Erasmus University, Rotterdam, the Netherlands.

References

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

[2] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall, *An analysis of dag-consistent distributed shared memory algorithms*, Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1996, Padua, Italy, 297–308.

[3] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, July 1995, Santa Barbara, CA, 207–216.

[4] R.D. Blumofe and C.E. Leiserson, *Space-efficient scheduling of multithreaded computations*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, May 1993, San Diego, CA, 362–371.

[5] R.D. Blumofe and C.E. Leiserson, *Scheduling multithreaded computations by work stealing*, in Proceedings of the 35th Annual Symposium on Foundations of Computer Science, November 1994, Santa Fe, NM, 356–368.

[6] I. Duff, R.G. Grimes, and J.G. Lewis, *Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*, CERFACS, October 1992, TR/PA/92/86.

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

[8] *Cilk-5.0 (Beta 1) Reference Manual*, Supercomputing Technology Group, Massachusetts Institute of Technology, March 1997, available on the World Wide Web at http://theory.lcs.mit.edu/~cilk.

Charles E. Leiserson and Aske Plaat are at the MIT Laboratory for Computer Science.