The Design and Implementation of Cilk 4

Matteo Frigo*
athena@theory.lcs.mit.edu

May 20, 1996

The implementation of Cilk 4 represents a radical point of departure from the previous versions of Cilk. First of all, the scheduler is now built on top of a shared-memory abstraction, rather then vice versa. In second place, we got rid of the old continuation-passing based scheduler, in favor of a lazy stack-based scheduler. These two main ideas, combined with some other tricks described later in the paper, allowed us to execute multithreaded programs almost as efficiently as the corresponding C programs. This abstract describes the implementation of Cilk 4, which is still work in progress and, for the most part, not yet published. We also have theoretical results proving that our scheduler is optimal from the point of view of execution time, and efficient in space utilization. However, the theory of Cilk 4 *is not* described here; the interested reader is referred to [2, 1].

In order to describe the implementation of Cilk, it sounds like a good idea to first explain what we are trying to implement. This is done in the section that follows.

1 Cilk's model of computation

A *Cilk multithreaded program* is a collection of thread definitions. Analogous to a C function definition, a thread definition is a block of serial code, possibly with conditional and looping constructions. Unlike a C function, however, a thread definition may contain various types of `spawn` and `sync` statements that allow the program to express concurrency, as follows. To specify parallelism, a thread may *spawn* child threads. A spawn is the parallel analogue of a procedure call, but, in the case of a spawn, the parent and child may execute concurrently. In addition a thread may *synchronize* with all its spawned children by suspending its execution until all children return. When the last child returns, it *enables* its parent to resume execution. Notice that a multithreaded program does not specify at what time or on what processor any given instruction is executed. An example of Cilk program is shown in Figure 1. The figure gives also the flavor of the Cilk language, which is essentially C plus the two keywords `spawn` and `sync`.

The execution of a multithreaded program can be viewed, in graph-theoretic terms, as a *multithreaded computation*. A multithreaded computation is composed of two structures: a "spawn tree" of threads and a dag of instructions. The *spawn tree* of threads is the parallel analogue of a call tree of C functions. The spawn tree is rooted at the "main" thread where algorithm execution begins, and in general, each spawned thread is a node in the spawn tree with the parent-child relationships defined by the spawn operations. The dag of instructions is the parallel analogue of a serial instruction stream. We think of the dag of instructions as being "embedded" in the spawn tree, since each executed instruction is part of a spawned thread. As illustrated in Figure 2, this embedding has the following properties. All of the instructions in any given thread are totally ordered by dag edges that we call *continue* edges. For each thread, except the root thread, its first instruction has exactly one incoming edge that we call a *spawn* edge, and this edge comes from an instruction (the spawning instruction) in the parent thread. For each thread, except the root thread, its last instruction has exactly one outgoing edge that we call a *return* edge, and this edge goes to an instruction (the synchronizing instruction) in the parent thread.

```
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;
     }
}
```

Figure 1: Example of Cilk program which computes the n-th Fibonacci number. The two calls to `fib(n-1)` and `fib(n-2)` can be done in parallel — this is why they are `spawn`ed. However both must complete before `sync` can to continue.

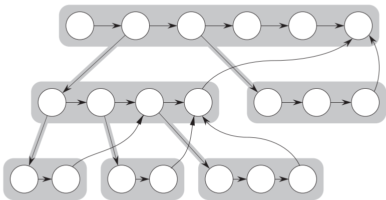

Figure 2: A fully strict multithreaded computation. Each node is an instruction, and each shaded region is a thread. The continue edges are horizontal, the spawn edges are shaded and downward pointing, and the return edges are curved and upward pointing.

In order to understand the implementation, the reader is urged to become familiar with the following three concepts.

Shared-memory model. In Cilk, threads can interact with each other trough a shared memory. For Cilk's shared memory we adopt a novel consistency model that we call *dag consistency*. An informal definition for dag consistency is the following:

Definition 1 *The shared memory of a multithreaded computation is dag-consistent if the following condition holds: whenever a read from the memory returns a value v, there is a sequential execution of the multithreaded dag in which that read also returns v.*

Notice that the definition allows different reads to return values corresponding to different sequential executions; in turn, this relaxation allows dag-consistency to be implemented efficiently on

* Joint work with Bobby Blumofe, Feng Ming Dong, Charles Leiserson, Keith Randall. This research was supported in part by the Advanced Research Projects Agency (ARPA) under Grants N00014-94-1-0985 and N00014-92-J-1310.

distributed-memory machines [2], yet providing enough semantics for many applications. It's also worth saying that this naïve definition is actually ill-defined for certain non-deterministic programs. For a formal definition of dag-consistency see [1].

Memory allocation: cactus stack. Some means of allocating memory must be provided by any non-useless implementation of shared memory. We have a general allocator in the style of `malloc()` and `free()`, but a less general mechanism, called *cactus stack*, is also useful by itself, and forms the core of the implementation of the runtime system.

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 incrementing and decrementing a stack pointer. The procedure views the stack as a linearly addressed space 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. The idea is that every procedure sees the stack *as if* it were the only active procedure in a sequential execution. In particular, this means that parts of the stack are shared among different procedures, and some other parts are private.

The Cilk scheduler. The Cilk scheduler is a greedy, randomized, work-stealing scheduler. The basic idea is the following: each thread has some memory associated with it, that we call *activation frame*. Whenever a thread spawns another thread, the state of the current thread is saved in the activation frame, which is in turn pushed onto the cactus stack (this is the same as the normal C function call mechanism). However, when a processor is idle, it chooses another processor at random, and tries to *steal* work from it. "Stealing work" means to take possession of the frame at the bottom of the stack, and continue the execution from the point of suspension. An important detail of the Cilk scheduler is the following: when a thread returns a value to its parent and enables it, the processor which is executing the child starts working on the parent, if nobody else is working on it. There might be, therefore, an implicit steal of the activation frame.

2 Implementation

We implemented Cilk 4 on a Sun SMP under Solaris 2[1]. Cilk consists of two parts: a type-checking preprocessor called `cilk2c`, which outputs C code from a Cilk input program, and a runtime system. From an high-level perspective, the runtime system provides the cactus stack, and the scheduler built on top of it. The scheduler also is responsible for maintaining the cactus stack dag-consistent. Some details are given in the following. In our implementation, the *processors* mentioned in the description of the scheduler are mapped into *processes*, forked at the beginning of the execution. The reason why we used normal processes instead of OS threads is twofold: first of all, we need to modify the memory map in arbitrary ways, in order to implement the cactus stack, and this is not allowed by the OS threads. In second place, OS threads have some limitations, most notably a fixed-size stack, which are problematic for a general-purpose runtime system.

The **cactus stack** consists of *pages* of memory, mapped into memory out of a globally-visible *backing store*, which, in our implementation, is a disk file. Pages of backing store can be mapped to arbitrary virtual memory addresses by means of the system call `mmap()`. The scheduler is responsible for the creation and destruction of branches of the cactus stack. Associated with each branch is a *page table* which maps memory addresses into pages of backing store. In the actual implementation the page table has a hierarchical tree structure.

Whenever a user program tries to access a page not yet mapped into memory, the hardware causes an exception, which is caught by the operating system. The OS, in turn, transfers control to our user-level signal handler, which maps the right page of backing store into memory, and then returns. In the course of this operation, the handler may discover that no page of backing store has yet been allocated for the address of cactus stack which caused the fault. In this situation, the handler allocates a new page and updates the page table accordingly.

The **scheduler** consists actually of two parts, that we will call *micro-* and *m*acro-scheduler. Here is the basic idea. For performance reasons, we would like our Cilk programs to run at C speed, at least most of the time. Unfortunately, unless we are willing to hack the C stack in a machine-dependent way, there is no way to do work-stealing on the C stack. We therefore chose a different approach. We maintain a separate stack (the cactus stack described earlier) as well as the ordinary C stack. For every Cilk procedure, the preprocessor `cilk2c` outputs two separate C functions, that we will name *fast* and *slow*. The fast version is just the C part of the input, with some additional code added by `cilk2c` in response to `spawn` and `sync`:

- Before `spawn`, `cilk2c` generates code to save the state on the cactus stack. The *state* includes all live variables, and information about the point where the thread was suspended.
- After `spawn`, `cilk2c` generates code to check if the thread was stolen. If so, the code stops executing immediately and returns to the runtime system.
- The instruction `sync` is actually a no-op in the fast version, since the fast version is scheduled in the ordinary C way, which guarantees that all `spawn`ed child threads have completed when a `sync` point is reached.

This light-weight scheduling activity is what we call *micro-scheduler*, and is implemented in such a way that the cost of `spawn` is less than three times the cost of an ordinary function call.

The slow version operates directly on the cactus stack, and has far more overheads. Moreover, it interacts with the work-stealing scheduler to perform the *macro-scheduling* activity. For the slow code, all variable accesses (like `n = 0`) are converted by `cilk2c` into cactus-stack accesses (something like `_frame->n = 0`); `spawn` is treated as in the fast case, but now `sync` generates an explicit check for termination of all child threads.

Overall, the macro-scheduling activity starts when the runtime system steals a frame from the cactus stack. At this point, a new branch of the cactus stack is created, and the slow code executed on the just-stolen frame. When a slow code spawns a new thread, however, the new thread can be executed on the C stack. Therefore, `spawn` is always spawn of the fast code, and the number of slow functions executed is always equal to the number of steals.

[1] A port to SGI SMPs is being completed in these days.

References

[1] 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*, Padua, Italy, June 1996. To appear.

[2] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Tenth International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, April 1996.