Cilk 5.3.1
Reference Manual

Supercomputing Technologies Group
MIT Laboratory for Computer Science

`http://supertech.lcs.mit.edu/cilk`

24 June 2000

Release Notes

Cilk-5.3.1

- Patched the use the unofficial gcc-2.96 that comes with redhat-7.0.
- `cilk2c`: stricter typechecking for inlet calls. Do not apply usual call conversions.
- Added support for `libhoard`.
- `cilk2c`: fixed bug regarding function redeclarations.

Cilk-5.3

- Cilk is now released under the GNU General Public License.
- The "cilk" command now automates all stages of compilation and linking. Use `cilk -O2 hello.cilk -o hello` to compile a simple program.
- Improved Linux support. `cilk2c` can now parse all Linux/glibc-2 include files.
- INCOMPATIBLE CHANGE: header file `cilk-compat.h` is no longer required and it is not provided.
- Fixed many bugs in `cilk2c`.
- Cilk now always runs on top of POSIX threads. The Cilk-5.2 distinction between threaded and process version has disappeared.
- The GNU linker is no longer required. Cilk should run on any system that supports gcc, GNU make, and POSIX threads.

Web page

The Cilk distribution, FAQ and further documentation are available at the Cilk web page:

<p align="center"><code>http://supertech.lcs.mit.edu/cilk</code></p>

Support

Please send questions and comments to

<p align="center"><code>cilk-support@lists.sourceforge.net</code></p>

Mailing lists

If you use Cilk, you may wish to be on the cilk-users mailing list. To join the cilk-users mailing list, look for the `cilk` project at `http://www.sourceforge.net`

To send email directly to the cilk-users mailing list, use

<p align="center"><code>cilk-users@lists.sourceforge.net</code></p>

Contents

Chapter 1

Introduction

This document describes Cilk 5.3.1, a language for multithreaded parallel programming based on ANSI C. Cilk is designed for general-purpose parallel programming, but is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style. Three world-class chess programs, ⋆Tech [19], ⋆Socrates [18], and Cilkchess, were written in Cilk by the Cilk group at MIT. Cilk provides an effective platform for programming dense and sparse numerical algorithms, such as matrix factorization [4] and N-body simulations. Unlike many other multithreaded programming systems, Cilk is algorithmic, in that the runtime system employs a scheduler that allows the performance of programs to be estimated accurately [5] based on abstract complexity measures.

1.1   What is included in Cilk 5.3.1?

This release of Cilk, which is available from `http://supertech.lcs.mit.edu/cilk`, includes the Cilk runtime system, the Cilk compiler, a collection of example programs, and this manual.

This version of Cilk is intended to run on Unix-like systems that support POSIX threads. In particular, Cilk runs on GNU systems on top of the Linux kernel, and it integrates nicely with the Linux development tools. Cilk 5.3.1 works on Linux/386, Linux/Alpha, Linux/IA-64, Solaris/SPARC, Irix/MIPS, and OSF/Alpha. Cilk should also run on other systems, provided that `gcc`, POSIX threads, and GNU `make` are available. If you would like help in porting Cilk to a new platform, please contact the Cilk developers by sending electronic mail to `cilk-support@lists.sourceforge.net`.

1.2   Background and goals

Cilk is an algorithmic multithreaded language. The philosophy behind Cilk is that a programmer should concentrate on structuring the 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. Thus, the Cilk runtime system takes care of details like load balancing, paging, and communication protocols. Unlike other multithreaded languages, however, Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance.

Cilk grew out of work in both theory and implementation. The theoretical input to Cilk comes from a study of scheduling multithreaded computations, and especially of the performance

of work-stealing, which provided a scheduling model that has since been the central theme of Cilk development. These results led to the development of a performance model that accurately predicts the efficiency of a Cilk program using two simple parameters: **work** and **critical-path length** [5, 7, 2]. More recent research has included **page faults** as a measure of locality [3, 4, 17].

The first implementation of Cilk was a direct descendent of PCM/Threaded-C, a C-based package which provided continuation-passing-style threads on Thinking Machines Corporation's Connection Machine Model CM-5 Supercomputer [20] and which used work-stealing as a general scheduling policy to improve the load balance and locality of the computation [14]. With the addition of a provably good scheduler and the incorporation of many other parallel programming features, the system was rechristened "Cilk-1." Among the versions of Cilk-1, notable was an adaptively parallel and fault-tolerant network-of-workstations implementation, called Cilk-NOW [2, 8].

The next release, Cilk-2, featured full typechecking, supported all of ANSI C in its C-language subset, and offered call-return semantics for writing multithreaded procedures. The runtime system was made more portable, and the base release included support for several architectures other than the CM-5.

Cilk-3 featured an implementation of dag-consistent distributed shared memory [4, 17]. With this addition of shared memory, Cilk could be applied to solve a much wider class of applications. Dag-consistency is a weak but nonetheless useful consistency model, and its relaxed semantics allows for an efficient, low overhead, software implementation.

In Cilk-4, the authors of Cilk changed their primary development platform from the CM-5 to the Sun Microsystems SPARC SMP. The compiler and runtime system were completely reimplemented, eliminating continuation passing as the basis of the scheduler, and instead embedding scheduling decisions directly into the compiled code. The overhead to spawn a parallel thread in Cilk-4 was typically less than 3 times the cost of an ordinary C procedure call, so Cilk-4 programs "scaled down" to run on one processor with nearly the efficiency of analogous C programs.

In Cilk-5, the runtime system was rewritten to be more flexible and portable than in Cilk-4. Cilk-5.0 could use operating system threads as well as processes to implement the individual Cilk "workers" that schedule Cilk threads. The Cilk-5.2 release included a debugging tool called the Nondeterminator [12, 9], which can help Cilk programmers to localize data-race bugs in their code. (The Nondeterminator is not included in the present Cilk-5.3 release.) With the current Cilk-5.3 release, Cilk is no longer a research prototype, but it attempts to be a real-world tool that can be used by programmers who are not necessarily parallel-processing experts. Cilk-5.3 is integrated with the `gcc` compiler, and the runtime system runs identically on all platforms. Many bugs in the Cilk compiler were fixed, and the runtime system has been simplified.

To date, prototype applications developed in Cilk include graphics rendering (ray tracing and radiosity), protein folding [24], backtracking search, N-body simulation, and dense and sparse matrix computations. Our largest application effort so far is a series of chess programs. One of our programs, $\star$Socrates [18], finished second place in the 1995 ICCA World Computer Chess Championship in Hong Kong running on the 1824-node Intel Paragon at Sandia National Laboratories in New Mexico. Our most recent program, Cilkchess, is the 1996 Open Dutch Computer Chess Champion, winning with 10 out of a possible 11 points. Cilkchess ran on a 12 processor 167 MHz UltraSPARC Enterprise 5000 Sun SMP with 1 GB of memory. The MIT Cilk team won First Prize, undefeated in all matches, in the ICFP'98 Programming Contest sponsored by the 1998

International Conference on Functional Programming.[2] We hope that by making Cilk available on the new generation of SMP's, other researchers will extend the range of problems that can be efficiently programmed in Cilk.

The four MIT Ph.D. theses [2, 17, 25, 13] contain more detailed descriptions of the foundation and history of Cilk. In addition, the minicourse [21], which was held as part of MIT's *6.046 Introduction to Algorithms* class during Spring 1998, provides a good introduction to the programming of multithreaded algorithms using a Cilk-like style.

1.3 About this manual

This manual is primarily intended for users who wish to write Cilk application programs. It also comprises information about implementation issues that can be useful when porting Cilk to a new system.

The manual is structured as follows. Chapter 2 is the programmer's guide, which introduces the features of the Cilk language. Chapter 3 contains the Cilk language reference manual. Chapter 4 contains the Cilk library reference manual. Chapter 5 describes internals of the Cilk 5.3.1 runtime system and compiler. This chapter should be useful if you are interested in Cilk internals or if you want to port Cilk to another platform.

---

[2]Cilk is not a functional language, but the contest was open to entries in any programming language.

Chapter 2

Programming in Cilk

This chapter introduces the Cilk language to programmers who are already reasonably familiar with C. Section 2.1 introduces the Cilk language through the example of computing Fibonacci numbers. Section 2.2 steps through the process of writing, compiling, and running simple Cilk programs. Section 2.3 describes how Cilk and C interact. Section 2.4 discusses storage allocation. Section 2.5 discusses shared memory. Section 2.6 describes the locking primitives that Cilk provides for atomicity. Section 2.7 treats some of Cilk's more advanced language features, including constructs for aborting already-spawned work and how to call Cilk procedures as library functions from C. Finally, Section 2.8 describes the computational model supported by Cilk's scheduler, and Section 2.9 describes Cilk's facilities for measuring performance.

Additional material on multithreaded programming using a Cilk-like model can be found in the minicourse notes [21]. The minicourse focuses on how recursive multithreaded algorithms can be designed and analyzed.

2.1 The Cilk language

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 the 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 simple. It consists of C with the addition of three 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. We call this C program the ***serial elison*** or ***C elision*** of the Cilk program. Cilk extends the semantics C in a natural way for parallel execution.

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 2.1(a), and Figure 2.1(b) shows a Cilk program that does the same computation in parallel.[1] Notice how

---

[1]This program is a terrible way to compute Fibonacci numbers, since it runs in exponential time while logarithmic-time methods are known [10, page 850].

```
                                          #include <stdlib.h>
                                          #include <stdio.h>
                                          #include <cilk.h>
    #include <stdlib.h>
    #include <stdio.h>                     cilk int fib (int n)
                                          {
    int fib (int n)                           if (n<2) return n;
    {                                         else
        if (n<2) return (n);                  {
        else                                      int x, y;
        {
            int x, y;                             x = spawn fib (n-1);
                                                  y = spawn fib (n-2);
            x = fib (n-1);
            y = fib (n-2);                        sync;

            return (x+y);                         return (x+y);
        }                                     }
    }                                     }

    int main (int argc, char *argv[])     cilk int main (int argc, char *argv[])
    {                                     {
        int n, result;                        int n, result;

        n = atoi(argv[1]);                    n = atoi(argv[1]);
        result = fib (n);                     result = spawn fib(n);

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

(b)

Figure 2.1: **(a)** A serial C program to compute the nth Fibonacci number. **(b)** A parallel Cilk program to compute the nth Fibonacci number.

similar the two programs look. In fact, the only differences between them are the inclusion of the library header file `cilk.h` and a few keywords: `cilk`, `spawn`, and `sync`.

The keyword `cilk` identifies a ***Cilk procedure***, which is the parallel version 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.
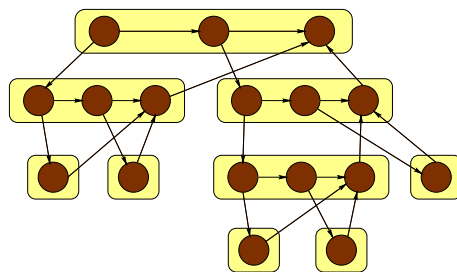
Figure 2.2: The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. All three types of edges are dependencies which constrain the order in which threads may be scheduled.

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, and not for all procedures currently executing. When all 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.

As an aid to programmers, Cilk inserts an implicit `sync` before every `return`, if it is not present already. As a consequence, a procedure never terminates while it has outstanding children.

The main procedure must be named `main`, as in C. Unlike C, however, Cilk insists that the return type of `main` be `int`. Since the `main` procedure must also be a Cilk procedure, it must be defined with the `cilk` keyword.

It is sometimes helpful to visualize a Cilk program execution as a directed acyclic graph, or ***dag***, as is illustrated in Figure 2.2. A Cilk program execution consists of a collection of ***procedures***,[2] 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` (either explicit or implicit) 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***

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. We shall discuss properties of the Cilk scheduler in Section 2.8.

---

[2]Technically, procedure ***instances***.

2.2 Compiling and running Cilk programs

In order to compile Cilk programs, Cilk 5.3.1 installs the `cilk` command, which is a special version of the `gcc` compiler. The `cilk` command understands that files with the `.cilk` extension are Cilk programs and acts accordingly.

`cilk` accepts the same arguments as the `gcc` compiler. For example, the source code for the Fibonacci example from Figure 2.1(b) can be found under the name `fib.cilk` in the `examples` directory of the Cilk distribution. After installing the distribution, you can type the command

```
$ cilk -O2 fib.cilk -o fib
```

to produce the `fib` executable. (You can also use your favorite `gcc` options, such as `-g`, `-Wall`, and so on.) To run the program, type:

```
> fib --nproc 4 30
```

this starts `fib` on 4 processors to compute the 30th Fibonacci number. At the end of the execution, you should see a printout similar to the following:

```
Result: 832040
```

During program development, it is useful to collect performance data of a Cilk program. The Cilk runtime system collects this information when a program is compiled with the flags `-cilk-profile` and `-cilk-critical-path`.

```
$ cilk -cilk-profile -cilk-critical-path -O2 fib.cilk -o fib
```

The flag `-cilk-profile` instructs Cilk to collect data about how much time each processor spends working, how many thread migrations occur, how much memory is allocated, etc. The flag `-cilk-critical-path` enables measurement of the critical path. (The critical path is the ideal execution time of your program on an infinite number of processors. For a precise definition of the critical path, see Section 2.8.) We distinguish the `-cilk-critical-path` option from `-cilk-profile` because critical-path measurement involves many calls to the timing routines, and it may slow down programs significantly. In contrast, `-cilk-profile` still provides useful profiling information without too much overhead.

Cilk program compiled with profiling support can be instructed to print performance information by using the `--stats` option. The command line

```
> fib --nproc 4 --stats 1 30
```

yields an output similar to the following:

```
Result: 832040

RUNTIME SYSTEM STATISTICS:

Wall-clock running time on 4 processors: 2.593270 s
Total work = 10.341069 s
Total work (accumulated) = 7.746886 s
Critical path = 779.588000 us
Parallelism = 9937.154003
```

After the output of the program itself, the runtime system reports several useful statistics collected during execution. Additional statistics can be obtained by setting the statistics level higher than 1.[3] The parallelism, which is the ratio of total work to critical-path length, is also printed so that you can gauge the scalability of your application. If you run an application on many fewer processors than the parallelism, then you should expect linear speedup.

The Cilk runtime system accepts the following standard options, which must be specified first on a Cilk program's command line:

`--help` List available options.

`--nproc` n Use n processors in the computation. If $n = 0$, use as many processors as are available. The parameter n must be at least 0 and be no greater than the machine size. The default is to use 1 processor.

`--stats` l Set statistic level to l. The higher the level, the more detailed the information. The default level is 0, which disables any printing of statistics. Level 1 prints the wall-clock running time, the total work, and the critical-path length. The exact set of statistics printed depends on the flags with which the runtime system and the application were compiled. The highest meaningful statistics level is currently 6.

`--no-stats` Equivalent to `-stats 0`.

`--stack` *size* Set a maximum frame stack size of *size* entries. The frame stack size limits the depth of recursion allowed in the program. The default depth is 32768.

`--` Force Cilk option parsing to stop. Subsequent options will be passed to the Cilk program unaltered.

These options must be specified before all user options, since they are processed by the Cilk runtime system and stripped away before the rest of the user options are passed to `main`.

Figure 2.3 illustrates the process by which a Cilk program is compiled. Cilk program files, which end with `.cilk` by convention, are first transformed to ordinary C code by the Cilk type-checking preprocessor `cilk2c`, producing `.c` files. The C code is then compiled using the `gcc` compiler and linked with the Cilk runtime system. You do not need to worry about this compilation process, though, since the `cilk` command takes care of everything.

2.3   Cilk and C

Cilk uses a source-to-source compiler, including a type-checking preprocessor that translates Cilk code into C code. You can incorporate C functions into Cilk code, but there are certain restrictions on how Cilk and C can interact.

The most basic rule governing Cilk and C is the following. Although it is permissible for a Cilk procedure to invoke a C function, C functions may not call or spawn Cilk procedures.

Thread-safe functions from system libraries also work when called from Cilk. The exact set of thread-safe functions is system dependent, but POSIX threads are now commmon enough that

---

[3]If critical path timing is turned on, the total work is measured in two different ways. The second, shown as `Total work (accumulated)`, is more accurate as it tries to compensate for the running time of the timer calls themselves.
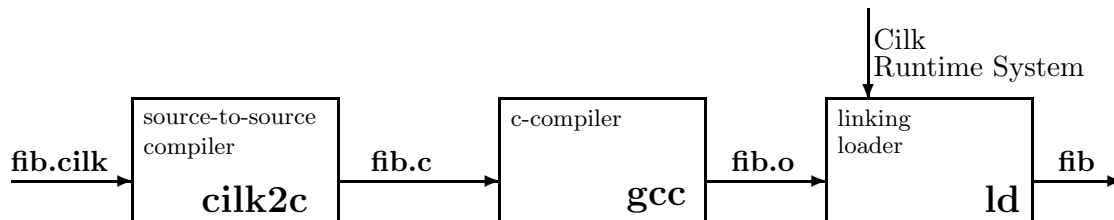
Figure 2.3: Compiling the Fibonacci program. Cilk code is passed through the `cilk2c` compiler, a type-checking preprocessor which generates C output. This output is then compiled with `gcc` and linked with the Cilk runtime library.

most libraries work fine with threads. One such library is `glibc-2.1`, which is installed on most GNU/Linux systems.

The function `alloca` is not thread-safe. See Section 4.2 for a description of a thread-safe replacement called `Cilk_alloca`.

Up to version 5.2, Cilk defined special versions of certain commonly-used functions. For example, Cilk-5.2 defined `Cilk_malloc`, which was guaranteed to work on all systems, as opposed to `malloc`. Cilk-5.2 provided a header file `cilk-compat.h`, which defined X to be `Cilk_X` for all redefined functions X. This cumbersome mechanism is now deprecated, and `cilk-compat.h` is no longer supported. Use `malloc` instead of `Cilk_malloc`.

2.4 Storage allocation

Like the C language on which it is based, Cilk provides two types of memory: stack and heap. Stack memory is allocated by the compiler for procedure-local variables and arrays, or you can allocate stack memory with the `Cilk_alloca()` library function available in `cilk-lib.h`. Stack memory is automatically deallocated when the Cilk procedure or C function that allocated the memory returns. Heap memory is allocated by the usual `malloc()` library function and deallocated with `free()`.

2.4.1 Stack memory

Cilk uses a ***cactus stack*** [23] for stack-allocated storage, such as the storage needed for procedure-local variables. As shown in Figure 2.4, 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 2.4.

Cactus stacks in Cilk have essentially the same limitations as ordinary C stacks [23]. 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
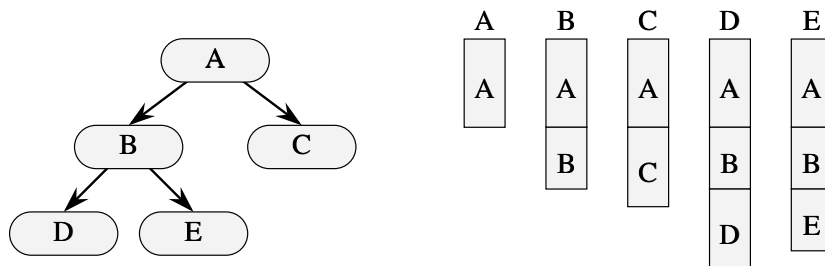
Figure 2.4: A cactus stack. Procedure A spawns B and C, and B spawns D and E. The left part of the figure shows the spawn tree, and the right part of the figure shows the view of the stack by the five procedures. (The stack grows downward.)

allocated on the cactus stack can only be safely passed to procedures below the allocation point in the call tree.

You can allocate `size` bytes of storage on the stack by calling the C library function `Cilk_alloca`:

```
ptr = Cilk_alloca(size);
```

Memory allocated by `Cilk_alloca` is freed when the procedure in which it was called returns.

In the current release, Cilk's version of `Cilk_alloca()` does not work properly when it is called from within a C function. Similarly, the C function `alloca()` does not work properly when called within a Cilk procedure.

2.4.2   Heap memory

To allocate heap storage, you call

```
ptr = malloc(size);
```

which allocates `size` bytes out of heap storage and returns a pointer to the allocated memory. The memory is not cleared. Heap storage persists until it is explicitly freed:

```
free(ptr);
```

where `ptr` is a pointer previously returned by a call to `malloc()`. Unlike storage allocated by `Cilk_alloca()`, a pointer to storage allocated by `malloc()` can be safely passed from a child procedure to its parent.

2.5   Shared memory

Cilk supports shared memory. Sharing occurs when a global variable is accessed by procedures operating in parallel, but sharing can also arise indirectly from the passing of pointers to spawned procedures, allowing more than one procedure to access the object addressed by the pointer. (Cilk supports the same semantics for pointer passing as C. See Section 2.4.) Updating shared objects in parallel can cause nondeterministic anomalies to arise, however. Consequently, it is important to understand the basic semantics of Cilk's shared-memory model.

```
cilk int foo (void)                    cilk int foo (void)
{                                      {
  int x = 0, y;                          int x = 0;

  spawn bar(&x);                         spawn bar(&x);
  y = x + 1;                             x = x + 1;
  sync;                                  sync;
  return (y);                            return (x);
}                                      }

cilk void bar (int *px)                cilk void bar (int *px)
{                                      {
  printf("%d", *px + 1);                 *px = *px + 1;
  return;                                return;
}                                      }
```

Figure 2.5: Passing the spawned procedure `bar` an argument consisting of a pointer to the variable `x` leads to the sharing of `x`.

Figure 2.6: Nondeterministic behavior may result from shared access to the variable `x` when `x` is updated.

Before delving into the semantics of shared memory, let us first see how sharing might occur in a Cilk program and what anomalies might arise. Figure 2.5 shows two Cilk procedures, `foo()` and `bar()`. In this example, `foo()` passes variable x to `bar()` by reference, and then it proceeds to read x before the `sync`.[4] Concurrently, `bar()` reads x through the pointer `px`. This way of sharing the value of x is safe, because the shared variable x is assigned in `foo()` before `bar()` is spawned, and no write accesses happen on x thereafter.

Figure 2.6 shows a slightly modified version of the Cilk procedures in Figure 2.5. Here, `foo()` passes the variable x to `bar()` by reference, but now it proceeds to modify x itself before the `sync`. Consequently, it is not clear what value `bar()` will see when it reads x through pointer `px`: the value at the time the variables were passed, the value after `foo()` has modified them, or something else. Conversely, it is not clear which value of x `foo()` will see. This situation is called a ***data race***, and it leads to nondeterministic behavior of the program. In most cases, nondeterminism of this sort is undesirable and probably represents a programming error.

The easiest way to deal with the anomalies of shared access is simply to avoid writing code in which one thread modifies a value that might be read or written in parallel by another thread. If you obey this rule, Cilk's shared-memory model guarantees that your program will deterministically produce the same values for all variables, no matter how the threads are scheduled. Determinacy is a goal to be strived for in parallel programming, although sometimes more parallelism can be obtained with a nondeterministic algorithm.

In some circumstances you may find you need to cope with the intricacies of sharing. In that case, you will need to understand something about the ***memory-consistency model*** of the machine on which you are running. For example, Figure 2.7 shows an example that could get you into trouble. If you reason through the logic of the program, you will see that the value printed for

---

[4]Actually, the `sync` statement in `foo()` is redundant, since the `return` statement implicitly imposes a `sync` immediately before the actual return.

```
int x = 0, y = 0, z = 0;

cilk void foo()
{
   x = 1;
   if (y==0) z++;
}

cilk void bar()
{
   y = 1;
   if (x==0) z++;
}

cilk int main ()
{
   spawn foo();
   spawn bar();
   sync;
   printf("z = %d\n", z);
   return 0;
}
```

Figure 2.7: An example of the subtleties of memory consistency.

z can be either 0 or 1, but not 2. Nevertheless, the value 2 may occasionally be printed on many contemporary SMP's. The reason is that some processors and compilers allow in foo the value x to be written after the value of y is read, since they refer to different locations. In a one-processor system, this change in the order of execution has no bearing on the correctness of the code, but on a multiprocessor anomalous results can occur, such as in this example, the printing of the value 2 for z.

As an aid to portability, Cilk provides a primitive Cilk_fence() that guarantees that all memory operations of a processor are committed before the next instruction is executed. Moreover, the compiler guarantees not to reorder operations across a Cilk_fence(). In the example from Figure 2.7, the printing of the anomalous value of 2 can be precluded by inserting a Cilk_fence() after the statement x = 1; in procedure foo and after the statement y = 1; in procedure bar. The fence forces each procedure to recognize the values written by the other in the proper fashion.

If the machine on which you run Cilk supports a given memory-consistency model, you are free to use it. You should be aware, however, that a different consistency model can cause your code not to port to another computer.

2.6 Locking

Cilk provides mutual exclusion locks that allow you to create atomic regions of code. A lock has type `Cilk_lockvar`. The two operations on locks are `Cilk_lock` to test a lock and block if it is already acquired, and `Cilk_unlock` to release a lock. Both functions take a single argument which is an object of type `Cilk_lockvar`.[5] Acquiring and releasing a Cilk lock has the memory consistency semantics of release consistency [16, p. 716]. The lock object must be initialized using `Cilk_lock_init()` before it is used. The region of code between a `Cilk_lock` statement and the corresponding `Cilk_unlock` statement is called a ***critical section***.

Here is an example:

```
#include <cilk-lib.h>
      :
      :
Cilk_lockvar mylock;
      :
      :
{
   Cilk_lock_init(mylock);
      :
   Cilk_lock(mylock); /* begin critical section */
      :
      :
   Cilk_unlock(mylock); /* end critical section */
}
```

Warning: **Locks must be initialized!** Make sure you initialize your locks before you use them.

2.7 Advanced features

This section introduces some of the advanced parallel programming features of Cilk. Section 2.7.1 introduces "inlets" as a means of incorporating a returned result of child procedures into a procedure frame in nonstandard ways. Cilk allows you to abort already-spawned work, a facility which is described in Section 2.7.2. Finally, Section 2.7.3 shows how a procedure can interact with Cilk's scheduler to test whether it is "synched" without actually executing a `sync`.

2.7.1 Inlets

Ordinarily, when a spawned procedure returns a value, it is simply stored into a variable in its parent's frame:

`x = spawn foo(y);`

---

[5]Prior to Cilk release 5.2, the argument to these functions was a pointer to an object of type `Cilk_lockvar`, not an object of type `Cilk_lockvar` itself. We changed the prototype of these functions so that an ampersand (`&`) is not normally needed when calling them.

```
cilk int fib (int n)
{
    int x = 0;
    inlet void summer (int result)
    {
        x += result;
        return;
    }

    if (n<2) return n;
    else {
        summer(spawn fib (n-1));
        summer(spawn fib (n-2));
        sync;
        return (x);
    }
}
```

Figure 2.8: Using an inlet to compute the nth Fibonnaci number.

Occasionally, you may find you want to incorporate the returned value into the parent's frame in a more complex way. Cilk provides an ***inlet*** feature for this purpose. Inlets are inspired in part by the inlet feature of TAM [11].

An inlet is essentially a C function internal to a Cilk procedure. Normally in Cilk, the spawning of a procedure must occur as a separate statement and not in an expression. An exception is made to this rule if the spawn is performed as an argument to an inlet. In this case, the procedure is spawned, and when it returns, the inlet is invoked. In the meantime, control of the parent procedure proceeds to the statement following the inlet.

Figure 2.8 illustrates how the `fib()` function might be coded using inlets. The inlet `summer()` is defined to take a returned value `result` and add it to the variable `x` in the frame of the procedure that does the spawning. All the variables of `fib()` are available within `summer()`, since it is an internal function of `fib()`.

Since an inlet is like a C function, it obeys the restrictions of C functions (see Section 2.3) in that it cannot contain `spawn` or `sync` statements. It is not entirely like a C function, because as we shall see, Cilk provides certain special statements that may only be executed inside of inlets. Because inlets cannot contain `spawn` or `sync` statements, they consist of a single Cilk thread.

It may happen that an inlet is operating on the variables of a procedure frame at the same time when the procedure itself or other inlets are also operating on those variables. Consequently, it is important to understand the effects of these interactions. *Cilk guarantees that the threads of a procedure instance, including its inlets, operate atomically with respect to one another.* In other words, you need not worry that variables in a frame are being updated by another thread while you are updating variables in the frame. This implicit atomicity of threads makes it fairly easy to reason about concurrency involving the inlets of a procedure instance without locking, declaring critical regions, or the like. On the other hand, Cilk guarantees nothing more than dag consistency in the interaction between two threads belonging to two different procedure instances. *Do not assume that threads of different procedure instances operate atomically with respect to each other.*

```
cilk int fib (int n)
{
    int x = 0;

    if (n<2) return n;
    else {
        x += spawn fib (n-1);
        x += spawn fib (n-2);
        sync;
        return (x);
    }
}
```

Figure 2.9: Using implicit inlets to compute the nth Fibonnaci number.

In principle, inlets can take multiple spawned arguments, but Cilk currently has the restriction that exactly one argument to an inlet may be spawned and that this argument must be the first argument. This limitation is easy to program around. If one wishes two separate arguments to be spawned, each spawn can be wrapped with its own inlet that incorporates its return value into the procedure frame.

Implicit inlets are used by Cilk to support the accumulation of a result returned by a spawned procedure. Figure 2.9 shows how the `fib()` function might be coded using implicit inlets. The accumulating assignment operator `+=` together with a `spawn` generates an implicit inlet definition and invocation by the Cilk compiler.

2.7.2 Aborting work

Sometimes, a procedure spawns off parallel work which it later discovers is unnecessary. This "speculative" work can be aborted in Cilk using the `abort` primitive inside an inlet. A common use of `abort` occurs during a parallel search, where many possibilities are searched in parallel. As soon as a solution is found by one of the searches, one wishes to abort any currently executing searches as soon as possible so as not to waste processor resources.

The `abort` statement, when executed inside an inlet, causes all of the already-spawned children of the procedure to terminate. Cilk does not guarantee that all children will be aborted instantly, however. For example, due to the vagaries of scheduling, a child may terminate normally after the `abort` statement is executed. Moreover, Cilk makes no promises regarding the return values of aborted children. If you use `abort`, it is your responsibility to handle the effects of premature termination.

Figure 2.10 illustrates how `abort` can be used in a program to solve the n-queens puzzle. The goal of the puzzle is to find a configuration of n queens on an $n \times n$ chessboard such that no queen attacks another (that is, no two queens occupy the same row, column, or diagonal). The idea is to use a backtracking search to explore the tree of possible configurations. As soon as a solution is found, the search is terminated using the `abort` primitive.

The Cilk procedure `nqueens` is called with three arguments: `config`, the current configuration of queens on the chess board; `n`, the total number of queens; and `i`, the number of queens which have already been placed on the chess board. The `spawn` in the `for` loop of `nqueens` causes searches

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cilk.h>
#include <cilk-lib.h>
#include <cilk-compat.h>

int safe(char *config, int i, int j)
{
    int r, s;

    for (r=0; r<i; r++) {
        s = config[r];
        if (j==s || i-r==j-s || i-r==s-j)
            return 0;
    }
    return 1;
}

cilk char *nqueens(char *config, int n, int i)
{
    char *new_config;
    char *done = NULL;
    int  j;

    inlet void catch(char *res)
    {
        if (res != NULL) {
            if (done == NULL)
                done = res;
            abort;
        }
    }

    if (i==n) {
        char *result;

        /* put this good solution in heap,
           and return a pointer to it */
        result = malloc(n*sizeof(char));
        memcpy(result, config, n*sizeof(char));
        return result;
    }
```

```c
    /* try each possible position for queen <i> */
    for (j=0; j<n; j++) {
        /* allocate a temporary array and
         * copy the config into it */
        new_config = Cilk_alloca((i + 1) * sizeof(char));
        memcpy(new_config, config, i*sizeof(char));
        if (safe(new_config, i, j)) {
            new_config[i] = j;
            catch(spawn nqueens(new_config, n, i+1));
        }

        if (done != NULL)
            break;
    }
    sync;

    return done;
}

cilk int main(int argc, char *argv[])
{
    int  n, i;
    char *config;
    char *result;

    if (argc < 2) {
        printf("%s: number of queens required\n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]);
    config = Cilk_alloca(n * sizeof(char));

    printf("running queens %d\n", n);
    result = spawn nqueens(config, n, 0);
    sync;

    if (result != NULL) {
        printf("Solution: ");
        for (i=0; i<n; i++)
            printf("%2d ", result[i]);
        printf("\n");
    }
    else
        printf("No solutions!\n");

    return 0;
}
```

Figure 2.10: Using an inlet with `abort` to find a solution to the n-queens puzzle.

```
cilk void baz(int n);

cilk void foo(int n)
{
    inlet void bar()
    {
        abort;
    }

    bar(spawn baz(17));
    spawn baz(28);
    sync;
}
```

Figure 2.11: Illustration of an anomaly with the use of `abort`.

of configurations starting from different columns of the current row to be spawned off in parallel. The C function `safe`, which checks whether the queen position at column `j` conflicts with other queens already set on the board, determines whether a given subsearch is spawned. Whenever one of the spawned subsearches completes, the inlet `catch` is executed, and the other subsearches are aborted if a solution has been found. Thus, when a solution has been discovered, the computation terminates.

When programming speculative computations using `abort`, there are some anomalies about which you should be aware. We have mentioned some of these anomalies: children may not be aborted instantly and control may be resumed at a `sync` without return values from aborted children having been set. A more subtle problem is sketched in Figure 2.11.

The procedure `foo` spawns two children: `baz(17)` and `baz(28)`. The question is, when `baz(17)` returns, and the inlet executes the `abort` statement, is `baz(28)` aborted? The answer is, it depends. If `baz(28)` has already been spawned but has not yet completed, then it is aborted. If `baz(28)` has not been spawned, then it will be spawned and will not be aborted. In other words, `abort` *only aborts extant children, not future children.* If you want to prevent future children from being spawned, you should set a flag in the inlet indicating that an `abort` has taken place, and then test that flag in the procedure before spawning a child. Since threads that belong to the same procedure execute atomically with respect to each other (including the thread comprising an inlet), no race conditions can occur whereby the procedure tests the flag, an `abort` occurs within an inlet, and then the procedure spawns a child.

2.7.3 Interacting with Cilk's scheduler

Cilk provides a simple means by which a procedure can determine whether it is "synched" without actually executing a `sync`, which might cause the procedure to stall. The variable `SYNCHED` has value 1 if the scheduler can guarantee that the procedure has no outstanding children and 0 otherwise.[6]

By testing `SYNCHED`, you can sometimes conserve space resources and obtain better locality than

---

[6]A child in not considered completed until all of its memory operations have completed. For versions of Cilk with a relaxed memory implementation, the completion of writes may occur well after the last instruction of the child is executed.

```
state1 = alloca(state_size);

/* fill in *state1 with data */

spawn foo(state1);

if (SYNCHED)
    state2 = state1;
else
    state2 = alloca(state_size);

/* fill in *state2 with data */

spawn bar(state2);

sync;
```

Figure 2.12: An illustration of the use of `SYNCHED` to save storage and enhance locality.

if you were completely oblivious to Cilk's scheduler. Figure 2.12 illustrates such a use of `SYNCHED`. The figure shows a Cilk procedure fragment in which two child procedures `foo` and `bar` are spawned. For each child, the procedure must allocate some auxiliary storage on the stack before performing the spawn. If the spawn of `foo` has completed executing before `bar` is spawned, however, then the space allocated for `foo` can be reused for `bar`. By checking the variable `SYNCHED`, the procedure detects this situation, and the space is reused. Otherwise, it allocates new space for `bar`.

The variable `SYNCHED` is actually more like a macro than an honest-to-goodness variable. The Cilk type-checking preprocessor [22] actually produces two "clones" of each Cilk procedure: a "fast" clone that executes common-case serial code, and a "slow" clone that worries about parallel communication. In the slow clone, which is rarely executed, `SYNCHED` inspects Cilk's internal state to determine if any children are outstanding. In the fast clone, which is almost always executed, however, the value of `SYNCHED` is the constant 1. Thus, for the fast clone, `cilk2c` translates any Cilk code that tests this variable into C code that tests against a constant. The optimizer in the C compiler recognizes that this test can be performed at compile time, and thus the test in the fast clone takes zero runtime.

2.8   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. In order to understand this guarantee better, this section surveys the major characteristics of Cilk's algorithmic model.

Recall that a Cilk program can be viewed as a dag, as was illustrated in Figure 2.2. To execute a Cilk program correctly, Cilk's underlying scheduler 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 [6]. Fortunately, every Cilk program generates a well-structured dag that can be scheduled efficiently [7].

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. (Here, we use the convention that the stack grows downward, and that items are pushed and popped from the "bottom" of the stack.) 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 to the one normally used by the worker.

Cilk's work-stealing scheduler executes any Cilk computation in nearly optimal time. From an abstract theoretical perspective, there are two fundamental limits to how fast a Cilk program could run. Let us denote with 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 with 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.[7] 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 [5], and the formula

$$T_P \approx T_1/P + T_\infty \tag{2.1}$$

is a good approximation of runtime. This performance model holds for Cilk programs that do not use locks. If locks are used, Cilk does not guarantee anything. (For example, Cilk does not detect deadlock situations.)

We can explore this performance model using the notion of **parallelism**, which is defined as $\overline{P} = T_1/T_\infty$. The parallelism is the average amount of work for every step along the critical path. Whenever $P \ll \overline{P}$, that is, the actual number of processors is much smaller than the 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 therefore the Cilk program is predicted 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's timing instrumentation (see Section 4.5) can measure these two quantities during a run of the program, no matter how many processors are used.

We can bound the amount of space used by a parallel Cilk execution in terms of its serial space. (We assume here that space is stack allocated and not heap allocated as is provided by the

---

[7]This abstract model of execution time ignores memory-hierarchy effects, but is nonetheless quite accurate [5].

C library function `malloc`.) Denote by S_P the 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 one runtime stack per processor. In fact, much less space may be required for many algorithms (see [3]), but the bound $S_P \leq S_1 P$ serves as a reasonable limit. This bound implies that, if a computation uses a certain amount of memory on one processor, it will use no more space per processor on average when run in parallel.

The algorithmic complexity measures of work and critical-path length—together with the fact that you can count on them when designing your program—justifies Cilk as an ***algorithmic*** multithreaded language. In addition, Cilk's scheduler also guarantees that the stack space used by a P-processor execution is never more than P times the stack space of an equivalent one-processor execution, and sometimes, it can be much less.

2.9   How to measure performance

The Cilk system can be configured to gather various runtime statistics as described in Section 2.2. Gathering those data imposes a performance penalty, however, as shown in Figure 2.13. There are two important reasons to measure execution time: to find out how fast the application is and to analyze what parts of an application are perhaps too slow. In the first case, the absolute time is of critical importance, whereas in the second case, the relative time is more important. Because measurements in Cilk perturb the program being measured, it is important to understand how to measure what you care about.

Cilk can be configured to provide the following measurements.

Performance Measurement: In order to measure execution time, Cilk should be used as follows.

1. *Compile the application* using the `cilk` command without the flag `-cilk-profile`.

2. *Run the program* without bothering about the runtime option `--stats` (see Section 2.2). Without the `-cilk-profile` flag, Cilk only collects the total work and the total running time. Higher levels of the `--stats` option do not provide significant insight.

Performance Analysis: When tuning a Cilk program for performance, various statistics can be useful to determine potential problems. These statistics include critical-path length, work, and number of steal attempts. For performance analysis, use Cilk as follows:

1. *Compile the application* using the `cilk` command with the flag `-cilk-profile`. If desired, also enable critical-path measurement with `-cilk-critical-path`. In the latter case, your program may be slowed down significantly, depending on the program and the availability of fast timers on your computer.

2. *Run the program* with your preferred level of the option `--stats` (see Section 2.2).

Figure 2.13 shows speedup measurements that were taken of the example programs distributed with Cilk-5.2. Do not expect these timings to be exactly reproducible in Cilk 5.3.1. We measured execution time on one processor T_1, execution time on eight processors T_8, work, critical-path length T_∞, and parallelism $\overline{P} = T_1/T_\infty$. To determine the overhead of the critical path measurements we repeated the experiment without critical-path mesurements. The execution times and the overhead due to critical-path measurements are reported in columns T_1', T_8' and T_1/T_1'.

	with critical path					w/o critical path		
Program	T_1	*Work*	T_∞	$\overline{P}$	T_8	T_1/T_8	T_1'	T_1/T_1'	T_8'
blockedmul	41.7	40.8	0.00114	35789	5.32	7.8	38.6	1.08	4.96
bucket	6.4	6.1	0.0318	192	1.02	6.3	6.2	1.036	1.02
cholesky	25.1	22.5	0.0709	317	3.68	6.8	14.9	1.68	2.32
cilksort	5.9	5.6	0.00503	1105	0.87	6.7	5.7	1.023	0.862
fft	13.0	12.5	0.00561	2228	1.92	6.8	11.2	1.16	1.85
fib	25.0	19.2	0.000120	160000	3.19	7.8	2.7	9.26	0.344
heat	63.3	63.2	0.191	331	8.32	7.6	63.0	1.0048	8.15
knapsack†	112.0	104.0	0.000212	490566	14.3	7.8	79.2	1.41	8.99
knary	53.0	43.0	2.15	20	20.2	2.6	12.7	4.17	9.19
lu	23.1	23.0	0.174	132	3.09	7.5	22.6	1.022	2.98
magic	6.1	5.5	0.0780	71	0.848	7.2	3.2	1.88	0.472
notempmul	40.4	39.8	0.0142	2803	4.96	8.0	37.5	1.077	4.71
plu	196.1	194.1	1.753	112	30.8	6.4	188.7	1.04	30.7
queens†	216.0	215.0	0.00156	137821	19.4	11.0	199.0	1.085	17.7
spacemul	39.3	38.9	0.000747	52075	4.91	8.0	37.1	1.059	4.73
strassen	4.2	4.1	0.154	27	0.767	5.5	4.2	1.0096	0.773
rectmul	5.0	5.0	0.000578	8599	0.638	7.8	4.6	1.082	0.606
barnes-hut	112.0	112.0	0.629	181	14.8	7.6	108.6	1.03	14.6

Figure 2.13: The performance of example Cilk programs. Time is measured in seconds. Measurements are only for the core algorithms. 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.

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, using two versions of Cilk-5 both compiled with `gcc` 2.8.1 and optimization level `-O3`.

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.

Two of the example programs, `queens` and `knapsack`, which are marked by a dagger (†) in the figure, are nondeterministic programs. Since the work of these programs depends on how they are scheduled, a superlinear speedup can occur. For these programs, the running time on one processor is not the same as the work performed by the computation. For the other (deterministic) programs, the measures of T_1 and work should be the same. In practice, differences occur, which are caused by a lack of timing accuracy. Because it reports work and critical-path length measurements, Cilk allows 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 figure, all programs exhibit good speedup. Even the complicated and irregular `Barnes-Hut` code achieves a speedup of 7.6 on 8 processors [25].

The right portion of Figure 2.13 compares execution times where the critical-path length mea-

surement is disabled with those where the critical-path length measurement is enabled. The performance gain from disabling the critical-path length measurement can be significant. On other architectures performance measurements with the critical-path length measurement enabled may be completely meaningless. Consequently, enabling the critical-path length measurements is only sensible for performance analysis purposes.

Chapter 3

Language Reference

This chapter defines the grammar for the Cilk language. The grammar rules are given as additions to the standard ISO C grammar (see [15, Appendix B]).

The typographic conventions used in this chapter are the same as [15]. In particular:

- Typewriter text (`like this`) denotes Cilk keywords or any other text that should be copied verbatim.

- Italicized text (*like this*) denotes grammatical classes (things like statements and declarations).

- Square brackets (like []) denote optional items.

3.1 Program structure

A Cilk program consists of ordinary ANSI C code plus definitions and declarations of Cilk procedures. A Cilk program contains one special Cilk procedure named `main`, which takes the standard C command line arguments and returns an integer, just like the C `main` function. At runtime, the `main` procedure is the initial procedure invoked from the runtime system, and the one from which all subsequent computation ensues.

Within a Cilk program, C code and C functions are used for sequential computation. C functions can be called from Cilk procedures, but Cilk procedures cannot be invoked from C functions.

3.2 Keywords

All C keywords are also keywords of Cilk. In addition, the following words have special meaning in Cilk, and thus cannot be used as ordinary identifiers in Cilk code:

`cilk, spawn, sync, inlet, abort, shared, private, SYNCHED`

The type qualifier `cilk` is used to declare Cilk procedures. Similarly, `inlet` is a type qualifier used to declare inlets (see Section 3.7).

Ordinary C code is permitted to use Cilk keywords as identifiers if one of the following conditions is true:

- The C code appears in a file whose extension is not `.cilk` or `.cilkh` (the conventional extensions for Cilk source and header files). The Cilk preprocessor treats files without these extensions as straight C. Thus, standard C header files included by a Cilk program may use Cilk reserved words as identifiers.

- the C code is preceded by `#pragma lang +C` and followed by `#pragma lang -C`. These directives tell the Cilk preprocessor to treat the code between them as straight C. This mechanism can be used when C code that contains Cilk keywords as ordinary identifiers is found within a Cilk program (`.cilk`) file.

For instance, the following C code uses `sync` as a variable name. It is bracketed by `#pragma lang` directives to prevent the preprocessor from issuing syntax errors.

```
#pragma lang +C
int sync;
sync = sync+1;
#pragma lang -C
```

3.3   Procedure definition and declaration

Cilk extends ANSI C with constructs to define procedures:

   declaration: `cilk` *type proc* (*arg-decl*$_1$, ..., *arg-decl*$_n$) *body*

The keyword `cilk` specifies a Cilk procedure definition. Argument declarations follow the ANSI C convention. When the body of the definition is the null statement (`;`), this form is taken to be a prototype declaration. An empty or `void` argument list specifies a procedure with no arguments. Cilk procedure bodies take the form of a C compound statement. Cilk procedure bodies may also contain Cilk statements, including procedure `spawn`'s, `sync`'s, `inlet` declarations, and other Cilk constructs.

   A Cilk procedure declaration can be used anywhere an ordinary C function declaration can be used. Cilk procedures must be `spawn`'ed; they cannot be called like ordinary C functions.

3.4   Spawning

The following Cilk statement forms can be used to spawn Cilk procedures to start parallel computations:

   statement: [*lhs op*] `spawn` *proc* (*arg*$_1$, ..., *arg*$_n$);

where *op* may be any of the following assignment operators:

   `=`   `*=`   `/=`   `%=`   `+=`   `-=`   `<<=`   `>>=`   `&=`   `^=`   `|=`

These forms are used to spawn Cilk procedures. The identifier *proc* must be defined or declared as a Cilk procedure with the correct number and types of arguments for *arg*$_1$, ..., *arg*$_n$. Normally, the left-hand side *lhs* of *op* must be present if and only if *proc* has a non-void return type, and *lhs* must have the same type as the return type of the procedure. Similarly, when *op* is `+=` or `-=` and the return type of *proc* is an integer, *lhs* may be a pointer in order to perform pointer arithmetic:

```
int  *p;
cilk int foo(...);

p += spawn foo(...);
```

A procedure may be spawned even if it is defined in a different source file, as long as a prototype for it is in scope.

Arguments passed to spawned procedures or threads can be any valid C expression.

Notice that the given syntax specifies a *statement*, not an *expression*. A syntax like

```
a = spawn foo() + spawn bar();
```

is invalid in Cilk.

A `spawn` statement can only appear within a Cilk procedure; it is invalid in C functions and in inlets.

Unlike C, no assignment conversion is done for the assignment operator in a spawn statement, and therefore code such as:

```
cilk float foo(void);
cilk void bar(void)
{
  int x;
  x = spawn foo();
  sync;
}
```

is invalid in Cilk. Instead, one must first assign the result to a variable of type `float` and then use a regular C assignment to convert this variable to an integer.[1]

3.5   Returning values

Within a Cilk procedure,

> statement: `return` [*expr*]`;`

can be used to communicate a return value to the spawning procedure, with the additional effect of terminating the execution of the current procedure.

3.6   Synchronization

Within the body of a Cilk procedure,

> statement: `sync;`

can be used to wait for all child procedures spawned before it in the current procedure to return. Typically, it is used to block the execution of the current procedure until all return values are available:

---

[1]We anticipate assignment conversion will be added to the semantics of `spawn` in a future release.

```
x = spawn foo (...);
y = spawn bar (...);
...
sync;
/* values of x, y available */
```

When the procedure resumes, execution starts at the point immediately after the `sync` statement.

The `sync` statement has additional meaning for dag-consistent shared memory. The `sync` statement ensures that all writes to shared memory performed by the `spawn`'s preceding the `sync` are seen by the code after the `sync`. This natural overloading of the `sync` statement for control flow synchronization and data synchronization makes dag-consistent shared memory easy to program and understand.

A `sync` statement may appear anywhere a statement is allowed in a Cilk procedure, including within the clause of an `if` statement and in other control constructs. A `sync` statement may not appear in an inlet definition.

3.7   Inlet definition

An inlet is a piece of code that specifies the actions to be done when a spawned procedure returns. Inlets must be defined in the declaration part of a procedure. An inlet executes atomically with respect to the spawning procedure and any other inlets that are declared in that procedure. Thus an inlet behaves as an atomic region that can update the parent frame.

An inlet should be defined in the declaration portion of a Cilk procedure:

declaration: `inlet` *type inlet-name* (*arg-decl*$_1$, ..., *arg-decl*$_n$) *body*

The keyword `inlet` specifies an inlet definition. Argument declarations follow the ANSI C convention.

Inlet bodies take the form of a C compound statement. The Cilk statement `abort` and the Cilk variable `SYNCHED` are allowed in inlet bodies. However, `spawn`, `sync`, and other `inlet` declarations are not allowed in inlet bodies. The scope of an inlet declaration is limited to the block in which it is declared.

3.8   Inlet invocation

Within the scope of an inlet declaration, the inlet can be invoked as:

statement: [*lhs =*] inlet (*arg*$_1$, ..., *arg*$_n$);

where *arg*$_1$ should be a spawn argument as:

`spawn` *proc* (*proc_arg*$_1$, ..., *proc_arg*$_m$)

Arguments *arg*$_2$, ..., *arg*$_n$ of the inlet and *proc_arg*$_1$, ..., *proc_arg*$_m$ of the procedure *proc* are first evaluated, and then *proc* is invoked as in the `spawn` statement (see Section 3.4). Control of the calling procedure may resume at the statement after the inlet invocation in parallel with the execution of *proc*. When *proc* completes execution, *inlet* is invoked with the result returned from the spawned procedure, if any, and with its other arguments *arg*$_2$, ..., *arg*$_n$. Note that *inlet* runs

in the lexical context of the procedure defining *inlet*, and thus it can reference automatic variables in its parent procedure.

At least one argument (that is, the first `spawn` argument) must be present in an inlet invocation. If *proc* has a non-void return type, the number and types of arg_1, ..., arg_n should match the number and types of *inlet*'s formal arguments. If *proc* has a void return type, the number and types of arg_2, ..., arg_n should match the number and types of *inlet*'s formal arguments. The return type of *inlet* should match the type of *lhs*. The value returned by *inlet* will be assigned to *lhs*. If *lhs* is not present, the return type of *inlet* must be `void`.

3.9   Aborting

Within the body of a Cilk inlet or Cilk procedure,

    statement: abort;

can be used to abort the execution of outstanding child procedures. The `abort` statement can be used to terminate prematurely useless work in a speculative execution. Any outstanding children may be stopped before completion, and any return values or side effects of these children may or may not be observed.

3.10   Cilk scheduler variables

The read-only Cilk scheduler variable `SYNCHED` allows a procedure to determine the progress of its spawned child procedures. Within a Cilk procedure or inlet, the variable `SYNCHED` can be used to test whether any outstanding child procedures exist. The variable `SYNCHED` has the value 1 if the scheduler can guarantee that there are no outstanding child procedures, and 0 otherwise. If `SYNCHED` has value 1, the scheduler also guarantees that all memory operations of all spawned children have completed as well. Because these memory operations may take a long time to complete in some Cilk implementations, `SYNCHED` may have value 0 even though all spawned children have finished executing.

The variable `SYNCHED` may only appear within a Cilk procedure or inlet and may not be used within C functions.

Chapter 4

Specification of Cilk Libraries

This chapter describes variables and functions that the Cilk system exports, and that may be used by Cilk programs. The Cilk library provides locks, debug support, timing functions, and replacement for C functions that are known not to be thread-safe.

To use the features described in this chapter, a program must include `cilk-lib.h`.

4.1   Global variables

```
extern int Cilk_active_size;
```

meaning: The variable `Cilk_active_size` is a read-only variable that specifies how many processes   Cilk is running on. The user may set this variable only by using the `--nproc` command-line option (see Section 2.2).

default: This variable defaults to 1 (one processor).

```
extern int Self;
```

meaning: The read-only variable `Self` specifies the current processor number. It is guaranteed to be between 0 and `Cilk_active_size`-1.

4.2   Shared memory support

For allocating stack memory dynamically, the following primitive is available:

```
void *Cilk_alloca(size_t size);
```

This primitive returns a pointer to a block of `size` bytes. Memory returned by `Cilk_alloca` is automatically freed at the end of the current Cilk procedure or C function, depending on where `Cilk_alloca` is called.

[**Implementation note:** The current implementation of `Cilk_alloca` is problematic. It works fine within Cilk procedures, but not inside C functions. Memory allocated with `Cilk_alloca` inside a C function is not automatically freed at the end of the function, and will persist until its parent Cilk procedure exits.]

4.3 Debug support

Cilk supports the following macro, analogous to `assert`.

`Cilk_assert(expression)`

meaning: `Cilk_assert()` is a macro that indicates `expression` is expected to be true at this point in the program. If `expression` is false (0), it displays `expression` on the standard error with its line number and file name, and then terminates the program and dumps core. Compiling with the preprocessor option `-DNDEBUG`, or with the preprocessor control statement `#define NDEBUG` in the beginning of the program, will stop assertions from being compiled into the program.

4.4 Locks

In Cilk, locks are sometimes required for mutual exclusion or to strengthen the memory model. Cilk locks are declared as type `Cilk_lockvar`. The following statements are available operations on lock variables:

`Cilk_lock_init(Cilk_lockvar l)` Initialize a lock variable. Must be called before any other lock operations are used on the lock.

`Cilk_lock(Cilk_lockvar l)` Attempt to acquire lock `l` and block if unable to do so.

`Cilk_unlock(Cilk_lockvar l)` Release lock `l`.

Cilk's locks are implemented to be as fast as is supported by the underlying hardware while still supporting release consistency [16, p. 716].

Along with the advantages of locks come some disadvantages. The most serious disadvantage of locks is that a deadlock might occur, if there is a cyclic dependence among locks. Deadlocks often arise from tricky nondeterministic programming bugs, and *the Cilk system contains no provisions for deadlock detection or avoidance.* Although this manual is not the place for a complete tutorial on deadlock avoidance, a few words would be prudent.

The simplest way to avoid deadlock in a Cilk program is to hold only one lock at a time. Unless a program contends for one lock while at the same time holding another lock, deadlock cannot occur.

If more than one lock needs to be held simultaneously, more complicated protocols are necessary. One way to avoid deadlocks is to define some global order on all locks in a program, and then acquire locks only in that order. For instance, if you need to lock several entries in a linked list simultaneously, always locking them in list order will guarantee deadlock freedom. Memory order (as defined by pointer comparison) can often serve as a convenient global order.

If a program cannot be arranged such that locks are always acquired in some global order, deadlock avoidance becomes harder. In general, it is a computationally intractable problem to decide whether a given program is deadlock-free. Even if the user can guarantee that his program is deadlock-free, Cilk may still deadlock on the user's code because of some additional scheduling constraints imposed by Cilk's scheduler. Additional constraints on the structure of lock-unlock pairs are required to make sure that all deadlock-free programs will execute in a deadlock-free

fashion with the Cilk scheduler. These additional constraints are as follows. First, each lock and its corresponding unlock must be in the same Cilk procedure. Thus, you cannot unlock a lock that your parent or child procedure locked. Second, every unlock must either follow its corresponding lock with no intervening `spawn` statements, or follow a `sync` with no intervening `spawn` statements. With these two additional constraints, Cilk will execute any deadlock-free program in a deadlock-free manner.

Locking can be dangerous even if deadlock is not a possibility. For instance, if a lock is used to make the update of a counter atomic, the user must guarantee that *every* access to that counter first acquires the lock. A single unguarded access to the counter can cause a *data race*, where the counter update does not appear atomic because some access to it is unguarded.

4.5   Timers

Cilk provides timers to measure the work and critical path of a computation as it runs. The overall work and critical-path length of a computation can be printed out automatically, as is described in Section 2.2. Sometimes, however, it is useful to measure work and critical path for subcomputations. This section describes such a timer interface. The interface is not particularly elegant right now, but with care, the work and critical path of subcomputations can be accurately measured.

The type `Cilk_time` contains a (hopefully platform-independent) Cilk time measure. The Cilk runtime system updates the variables `Cilk_user_critical_path` and `Cilk_user_work` at thread boundaries when the user code is compiled with the `-cilk-critical-path` flag. These variables are stored using Cilk-internal time units. The function `Cilk_time_to_sec()` converts the Cilk-internal units into seconds.

The normal way to determine the work or critical path of a subcomputation is to take a reading of `Cilk_user_critical_path` or `Cilk_user_work` before the subcomputation, take another reading after the subcomputation, and subtract the first reading from the second. Because these variables are only updated at thread boundaries, however, reliable timings can only be obtained in general if they are read at the beginning of a procedure or immediately after a `sync`. Reading them at other times may yield innaccurate values.

In addition to work and critical-path lengths, the function `Cilk_get_wall_time()` returns the current wall-clock time in Cilk-internal wall-clock units. The wall-clock units are not necessarily the same as the work and critical-path units. The function `Cilk_wall_time_to_sec()` converts the Cilk-internal units into seconds.

Here is an example usage.

```
Cilk_time tm_begin, tm_elapsed;
Cilk_time wk_begin, wk_elapsed;
Cilk_time cp_begin, cp_elapsed;

/* Timing. "Start" timers */
sync;
cp_begin = Cilk_user_critical_path;
wk_begin = Cilk_user_work;
tm_begin = Cilk_get_wall_time();
```

```
spawn cilksort(array, tmp, size);
sync;

/* Timing. "Stop" timers */
tm_elapsed = Cilk_get_wall_time() - tm_begin;
wk_elapsed = Cilk_user_work - wk_begin;
cp_elapsed = Cilk_user_critical_path - cp_begin;

printf("\nCilk Example: cilksort\n");
printf("          running on %d processor(s)\n\n", Cilk_active_size);
printf("options: number of elements = %ld\n\n", size);
printf("Running time  = %4f s\n", Cilk_wall_time_to_sec(tm_elapsed));
printf("Work          = %4f s\n", Cilk_time_to_sec(wk_elapsed));
printf("Critical path = %4f s\n\n", Cilk_time_to_sec(cp_elapsed));
```

Chapter 5

How Cilk Works

This section outlines the basic runtime assumptions and compilation schemes for Cilk procedures, which may be useful to those wishing to understand more about Cilk implementation or the C code generated by the `cilk2c` compiler.

We first describe how a Cilk program is scheduled on one processor (nanoscheduling), and then describe how a Cilk program is scheduled on multiple processors. Cilk currently has no job scheduler (the scheduling of multiple simultaneous Cilk programs).

5.1 Nanoscheduling

The nanoscheduler's job is to schedule procedures within a single processor. It is very important that the nanoscheduler make scheduling decisions quickly, so the nanoscheduler is "compiled" into the source code by `cilk2c`. The nanoscheduler's algorithm is very simple: it executes a procedure and its subprocedures in exactly the same order as they would execute in C. This schedule guarantees that on one processor, when no microscheduling is needed, the Cilk code executes in the same order as the C code.

The nanoscheduler is very easy to implement. The `cilk2c` compiler translates each Cilk procedure into a C procedure with the same arguments and return value. Each spawn is translated into its equivalent C function call. Each sync is translated into a no-operation, because all children would have already completed by the time the sync point is reached.

In order to enable the use of our microscheduler, we must add some code to the nanoscheduler to keep track of the current scheduling state. The nanoscheduler uses a deque of frames for this purpose. A Cilk *frame* is a data structure which can hold the state of a procedure, and is analogous to a C activation frame. A *deque* is a doubly-ended queue, which can be thought of as a stack from the nanoscheduler's perspective. The deque is tightly coupled to the C stack, with a one-to-one correspondence between the frames on the deque and the activation frames on the C stack.[1]

The nanoscheduler performs the following additional operations to keep track of the state of the computation. When a procedure is first called, it initializes a frame to hold its saved state, and pushes this frame on the bottom of the frame deque. When a spawn occurs, the state of the

---

[1]We maintain a separate deque instead of using the native C stack to maintain portability. Someday, we hope to merge the frame deque and the native C stack, which will further reduce the overhead of the nanoscheduler.

```
struct _fib_frame {
  StackFrame header;
  struct {int n;} scope0;
  struct {int x;int y;} scope1;
};
int fib(int n)
{
  struct _fib_frame *_frame;
  _INIT_FRAME(_frame,sizeof(struct _fib_frame),_fib_sig);
  {
    if (n < 2)
      {_BEFORE_RETURN_FAST();return (n);}
    else {
      int x;int y;
      { _frame->header.entry=1;
        _frame->scope0.n=n;
        x=fib(n-1);
        _XPOP_FRAME_RESULT(_frame,0,x);
      }
      { _frame->header.entry=2;
        _frame->scope1.x=x;
        y=fib(n-2);
        _XPOP_FRAME_RESULT(_frame,0,y);
      }
      /* sync */;
      {_BEFORE_RETURN_FAST();return (x+y);}
    }
  }
}
```

Figure 5.1: Nanoscheduled version of `fib`.

current procedure is saved in this frame. Finally, when the procedure completes and returns to its parent, its frame is popped.

Figure 5.1 shows the nanoscheduled version of the Fibonacci example from Figure 2.1(b). `cilk2c` generates a frame for the `fib` procedure to hold all of its state. The state of `fib` includes its argument `n` and its two local variables, `x` and `y`. Additional state, such as the procedure ID and program counter, are stored in the header.

The procedure fib has one additional local variable, **_frame**, which holds a pointer to the procedure's frame. The frame is initialized and pushed onto the deque with the **_INIT_FRAME** macro. For each spawn, the state of the procedure is saved into the frame. The state that is saved includes the entry number (representing the program counter), and each live, dirty variable. For instance, the first spawn, identified by entry number 1, saves the entry number and the argument `n` into the frame. The spawned procedure is then called with a normal C function call. After the function call returns, execution of the parent resumes after the spawn. The second spawn is the same as the first, except it has entry number 2, and the only live, dirty variable is `x`.

The nanoscheduler must check to see whether the procedure has been migrated after each spawn call. This check is done in the **_XPOP_FRAME_RESULT** macro by checking if the current frame

is still in the frame deque. If the frame is not in the deque, then it has been migrated, in which case `_XPOP_FRAME_RESULT` returns immediately with a dummy return value. This immediate return feature quickly cleans up the C stack when the frame deque becomes empty. Otherwise, the procedure can be resumed after the spawn call in normal C fashion.

Finally, when a procedure returns, the nanoscheduler pops its frame off the deque and frees it using the `_BEFORE_RETURN_FAST` macro.

Note that during nanoscheduling, the state information written to the frame deque is never used. The nanoscheduler is simply recording this information for use by the microscheduler, which is described in the next section.

5.2   Microscheduling

The microscheduler's job is to schedule procedures across a fixed set of processors. The microscheduler is implemented as a randomized work-stealing scheduler. Specifically, when a processor runs out of work, it becomes a ***thief*** and steals work from a ***victim*** processor chosen uniformly at random. When it finds a victim with some frames in its deque, it takes the topmost frame (the least recently pushed frame) on the victim's deque and places it in its own deque. It then gives the corresponding procedure to the nanoscheduler to execute.

The procedure given to the nanoscheduler is a different version of the stolen procedure, however. We call this version the "slow" version of the procedure, because it has a higher overhead than the normal nanoscheduled routine. Because the slow version is invoked from a generic scheduling routine, its interface is standard. Instead of receiving its arguments as C arguments, it receives a pointer to its frame as its only argument, from which it can extract its arguments and local state. It also produces a return value using the macro `_SET_RESULT` instead of returning a result normally. Finally, because this procedure may have outstanding children on another processor, it may suspend at a synchronization point.

Figure 5.2 shows the microscheduled version of the Fibonacci example from Figure 2.1(b). The microscheduled version takes a pointer to the frame as its only argument, and returns void. The first thing the microscheduled function does is to restore its state, first restoring the program counter using the `switch` statement, and then restoring local variables using the state-restoring code at each `sync` label.

For each return in the Cilk code, instead of returning the value as in C, the slow version uses `_SET_RESULT` to record the return value in a temporary area and then returns a void value. The runtime system can then read the return value from this temporary area and store it in the appropriate final destination.

At each synchronization point, the slow version needs to save its state and then check whether any return values are still outstanding. It does this with the `_SYNC` macro, which returns non-zero if any children are still outstanding. If there are still children outstanding, the slow version returns to the scheduler, where it is suspended by the runtime system.

Since slow versions of procedures are created only during a steal, they are always topmost in a processor's frame deque. All other procedures in the frame deque are the standard, fast, nanoscheduled versions. Therefore, as long as the number of steals is small, most of the computation is performed using the nanoscheduled routines.

```
static void _fib_slow(struct _fib_frame *_frame)
{
  int n;
  switch (_frame->header.entry) {
  case 1: goto _sync1;
  case 2: goto _sync2;
  case 3: goto _sync3;
  }
  n=_frame->scope0.n;
  {
    if (n < 2)
      {_SET_RESULT((n));_BEFORE_RETURN_SLOW();return;}
    else {
      { int _temp0;
        _frame->header.entry=1;
        _frame->scope0.n=n;
        _temp0=fib(n-1);
        _XPOP_FRAME_RESULT(_frame,/* return nothing */,_temp0);
        _frame->scope1.x=_temp0;
        if (0) { _sync1:; n=_frame->scope0.n; }
      }
      { int _temp1;
        _frame->header.entry=2;
        _temp1=fib(n-2);
        _XPOP_FRAME_RESULT(_frame,/* return nothing */,_temp1);
        _frame->scope1.y=_temp1;
        if (0) { _sync2:; }
      }
      { _frame->header.entry=3;
        if (_SYNC) {
          return;
          _sync3:;
        }
      }
      { _SET_RESULT((_frame->scope1.x+_frame->scope1.y));
        _BEFORE_RETURN_SLOW();return;
      }
    }
  }
}
```

Figure 5.2: Microscheduled version of `fib`.

5.3   Overheads

The overhead of the nanoscheduler, as compared to a serial C version, comes from five factors:

- A procedure's frame needs to be allocated, initialized, and pushed on the deque at the start of each procedure. These operations can be performed in less than a dozen assembly instructions.

- A procedure's state needs to be saved before each spawn. The entry number and each live, dirty variable must be saved. Note that some memory synchronization operations are also required at this point for architectures which do not implement sequential consistency (see Section 5.4 for details).

- A procedure must check whether its frame has been stolen after each spawn. This check requires only two reads, a compare, and a branch. Again, memory synchronization operations are required at this point for architectures which do not implement sequential consistency.

- On return, a procedure must free its frame, which takes only a few instructions.

- One extra variable is needed to hold the frame pointer. This extra variable imposes a small amount of additional register pressure on the compiler's register allocator.

This overhead is very small in practice. For instance, the Cilk Fibonacci program runs only 3 times slower than the sequential C program on one processor. For programs with larger thread lengths, this overhead would be minimal.

5.4   Interactions with weak memory models

In general, the Cilk 5.3.1 scheduler will work for any reasonable consistency model supported by modern SMPs. The Cilk scheduler requires stronger consistency in the three following areas:

- The scheduler requires that obtaining a lock on a data structure guarantees atomic access to that data structure.

- When a frame is pushed on the deque, the scheduler requires that all writes must be globally visible before the tail pointer (the end of deque pointer) is incremented. In systems where writes can be reordered, a memory barrier of some sort must be inserted before the pointer increment. Writes remain ordered on our current platform (UltraSPARC SMP), so no barrier is needed for our implementation.

- When a frame is popped from the deque, the scheduler requires that the write of the tail pointer be globally visible before the read of the exception pointer is performed. In systems where reads can be performed before preceeding writes are globally visible, a memory barrier of some sort must be inserted before the read of the exception pointer. Our UltraSPARC SMP implementation requires a "StoreLoad" barrier at this point.

Note that the last two requirements are simply optimizations to avoid the cost of a lock acquire and release on each push and pop of a frame. In the worst case, the last two requirements can be implemented using a lock acquire and release.

Appendix A

People

The Cilk project is led by Prof. Charles E. Leiserson of the MIT Laboratory for Computer Science and has been funded in part by DARPA Grant F30602-97-1-0270.

The current maintainer of Cilk is Matteo Frigo (`athena@fftw.org`), who is responsible for the Cilk 5.3.1 release.

Cilk 5.3.1 is the product of the effort of many people over many years. The following people contributed code, example programs, and/or documentation to the Cilk 5.3.1 system:

Guang-Ien Cheng
Don Dailey
Mingdong Feng
Charles E. Leiserson
Phil Lisiecki
Alberto Medina
Harald Prokop
Keith H. Randall
Bin Song
Andy Stark
Volker Strumpen

All those who have worked on previous versions of Cilk:

Robert D. Blumofe
Michael Halbherr
Christopher F. Joerg
Bradley C. Kuszmaul
Howard Lu
Robert Miller
Aske Plaat
Richard Tauriello
Daricha Techopitayakul
Yuli Zhou

Finally, thanks to those who have spent valuable time with Cilk and contributed with their experience:

Arvind
Reid Barton
Ching Law
John Litvin
Igor Lyubashevskiy
Rolf Riesen
Andy Shaw
Mike Stupak
Adrian M. Soviani
Svetoslav Tzvetkov

Appendix B

Copyright and Disclaimers

Bibliography

[1] Robert D. Blumofe. Managing storage for multithreaded computations. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1992. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-552.

[2] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[3] Robert D. Blumofe, Matteo Frigo, Christopher 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)*, Padua, Italy, June 1996.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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 (HPDC)*, pages 96–105, San Francisco, California, August 1994.

[9] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM*

Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, Mexico, June 28–July 2 1998.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[11] 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 (ASPLOS)*, pages 164–175, Santa Clara, California, April 1991.

[12] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. Submitted for publication. Available at `ftp://theory.lcs.mit.edu/pub/cilk/spbags.ps.gz`, January 1997.

[13] Matteo Frigo. *Portable High-Performance Programs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[14] Michael Halbherr, Yuli Zhou, and Christopher 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.

[15] Samuel P. Harbison and Guy L. Steele Jr. *C, A Reference Manual*. Tartan, Inc., 1995.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[17] 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.

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

[19] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD 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`.

[20] 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 (SPAA)*, pages 272–285, San Diego, California, June 1992.

[21] Charles E. Leiserson and Harald Prokop. A minicourse on multithreaded programming. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`, 1998.

[22] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[23] 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.

[24] 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.

[25] Keith H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.