Cilk 5.2
Reference Manual

Supercomputing Technologies Group
MIT Laboratory for Computer Science

`http://theory.lcs.mit.edu/~cilk`

July 21, 1998

Cilk is a trademark of the Massachusetts Institute of Technology.
The Cilk project is supported in part by DARPA Grant F30602-97-1-0270.
This manual is Copyright © 1998 Massachusetts Institute of Technology.

Release Notes

Cilk-5.2 (Beta 1)

- New Nondeterminator version debugs data races in programs with locks.
- New Library version allows the Cilk runtime system to be started from within a C program.
- Added support for Digital UNIX on Alpha.
- Lock functions `Cilk_lock`, `Cilk_unlock`, and `Cilk_lock_init` were changed to take a `Cilk_lockvar` instead of a pointer to a `Cilk_lockvar`.

Web page

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

`http://theory.lcs.mit.edu/~cilk`

Support

Please send questions and comments to

`cilk-support@supertech.lcs.mit.edu`

Mailing lists

If you use Cilk, you may wish to be on the cilk-users mailing list. To join the cilk-users mailing list, send email to

`cilk-users-request@supertech.lcs.mit.edu`

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

`cilk-users@supertech.lcs.mit.edu`

Contents

Release Notes **i**

Part I — Tutorial 1

1 Introduction **3**
- 1.1 What is included in Cilk 5.2? . . . 3
- 1.2 Background and goals . . . 3
- 1.3 About this manual . . . 5

2 Programming in Cilk **6**
- 2.1 The Cilk language . . . 6
- 2.2 Compiling and running Cilk programs . . . 9
- 2.3 Cilk and C . . . 11
- 2.4 Storage allocation . . . 12
  - 2.4.1 Stack memory . . . 12
  - 2.4.2 Heap memory . . . 13
- 2.5 Shared memory . . . 13
- 2.6 Locking . . . 16
- 2.7 Advanced features . . . 17
  - 2.7.1 Inlets . . . 17
  - 2.7.2 Aborting work . . . 19
  - 2.7.3 Interacting with Cilk's scheduler . . . 21
  - 2.7.4 Library version . . . 22
- 2.8 Debugging . . . 23
- 2.9 The Cilk model of multithreaded computation . . . 25
- 2.10 How to measure performance . . . 27

3 Example Programs **30**

Part II — Implementation Notes 43

4 Cilk Runtime System Implementor's Guide **45**
4.1 How Cilk 5.2 views memory 45
4.2 Locks in Cilk 47
4.3 Interaction with weak memory models 47
4.4 Architecture-specific issues 48
4.5 The Nondeterminator 49

5 How Cilk Works **51**
5.1 Nanoscheduling 51
5.2 Microscheduling 53
5.3 Overheads 55
5.4 Interactions with weak memory models 56

Part III — User's Guide **57**

6 Language Reference **59**
6.1 Program structure 59
6.2 Keywords 59
6.3 Procedure definition and declaration 60
6.4 Spawning 61
6.5 Returning values 62
6.6 Synchronization 62
6.7 Inlet definition 62
6.8 Inlet invocation 63
6.9 Aborting 63
6.10 Cilk scheduler variables 64
6.11 Shared memory 64
6.12 Locks 65
6.13 Nonstandard features 66
6.13.1 Shared and private variables 66
6.13.2 Timers 67
6.13.3 OS-thread runtime system 68
6.14 Open issues 69
6.14.1 C libraries 69
6.14.2 File and socket I/O 69
6.14.3 Unimplemented features 70

7 The Cilk Compiler **71**

8 Specification of Cilk Libraries **74**
8.1 Global variables 74
8.2 Library functions 74
8.2.1 Shared memory support 75

8.2.2 Random number generator 75
8.2.3 Default libc routines 75
8.2.4 Library version 76
8.3 Macros 76
8.4 Locks 77
8.5 Compatibility mode 77

9 The Nondeterminator **78**
9.1 The Nondeterminator's race reports 78
9.2 Memory allocation in the Nondeterminator 79
9.3 The Brelly algorithm 79
9.4 Limitations of the Nondeterminator 80
9.5 Fake locks and memory publishing 81
9.6 The Nondeterminator's compile-time warnings 82
9.7 Inlets 83
9.8 The Nondeterminator's `cilk2c` options 84
9.9 The Nondeterminator's runtime options 84
9.10 The Nondeterminator's library functions / macros 84
9.11 Pragmas and attributes 85

A Installation Instructions **87**

B FAQ: Frequently Asked Questions **90**

C People **94**

D Copyright and Disclaimers **96**

Part I — Tutorial

Chapter 1

Introduction

This document describes Cilk 5.2, a language for multithreaded parallel programming based on ANSI C. Cilk is designed for general-purpose parallel programming, but it is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style. Our group has developed three world-class chess programs, ⋆Tech [21], ⋆Socrates [19], and Cilkchess. Cilk provides an effective platform for programming dense and sparse numerical algorithms, such as matrix factorization [5] and N-body simulations, and we are working on other types of applications. 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 [6] based on abstract complexity measures.

1.1 What is included in Cilk 5.2?

This base release, which is available from `http://theory.lcs.mit.edu/~cilk`, includes the Cilk runtime system, the `cilk2c` compiler, a collection of example programs, and this manual. The full release includes the base release and the Nondeterminator debugging tool. The Cilk runtime system included in this release runs on Sun SPARC machines running Solaris, Silicon Graphics machines running Irix, and x86 machines running Linux. Earlier versions of Cilk have been ported to other platforms—including PVM, the Intel Paragon, and networks of workstations—but these ports are not available for this release. The current release works only on SMP's. We are currently working on a version for distributed memory multiprocessors. If you would like help in porting Cilk to a new platform, please contact the Cilk developers by sending electronic mail to `cilk-support@supertech.lcs.mit.edu`.

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*** [6, 8, 3]. More recent research has included ***page faults*** as a measure of locality [4, 5, 20].

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 [22] and which used work-stealing as a general scheduling policy to improve the load balance and locality of the computation [16]. 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 platforms that supported Cilk-1, notable was an adaptively parallel and fault-tolerant network-of-workstations implementation, called Cilk-NOW [3, 9].

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 [5, 20]. 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 is typically less than 3 times the cost of an ordinary C procedure call, so Cilk programs "scale down" to run on one processor with nearly the efficiency of analogous C programs.

In Cilk-5, the runtime system has been rewritten to be more flexible and portable than in Cilk-4. Cilk-5 can use operating system threads as well as processes to implement the individual Cilk "workers" that schedule Cilk threads. Using operating system threads solves some of the I/O naming problems that are present in the process version. Also included in this release is a debugging tool called the Nondeterminator [14, 10], which can help Cilk programmers to localize data-race bugs in their code.

To date, prototype applications developed in Cilk include graphics rendering (ray tracing and radiosity), protein folding [26], backtracking search, N-body simulation, and dense and sparse matrix computations. Our largest application effort is a series of chess programs. One of our programs, ⋆Socrates [19], 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 of 11 points. Cilkchess ran on a 12 processor 167 MHz UltraSPARC Enterprise 5000 Sun SMP with 1 GB of memory. (Go to `http://theory.lcs.mit.edu/~cilk/tournament.html` to replay the games.) We hope that by making Cilk available on the new generation of SMP's, others will extend the range of problems that can be efficiently programmed in Cilk.

The three MIT Ph.D. theses [3, 20, 27] contain more detailed descriptions of the foundation and history of Cilk. In addition, the minicourse [23], 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 are useful to port Cilk to a new system. The manual is structured into three parts: (I) Tutorial, (II) Implementation Notes, and (III) User's Guide.

Part I covers introductory material on Cilk. Chapter 2 is a programmer's guide that introduces the features of the Cilk language. Chapter 3 describes a variety of programs that are included in the Cilk distribution. These programs provide good examples for learning to program in Cilk.

Chapters 4 and 5 of Part II describe internals of the Cilk 5.2 runtime system and compiler. This part should be useful if you want to educate yourself about Cilk internals or if you want to port Cilk to another platform.

The User's Guide constitutes Part III, which contains chapters on the Cilk language, the Cilk compiler, the Cilk library, and the Nondeterminator race detection tool.

Installation instructions for Cilk can be found in Appendix A. Appendix B offers answers to some frequently asked questions about Cilk.

Chapter 2

Programming in Cilk

This chapter introduces the Cilk language to programmers who are already reasonably familiar with C. Our descriptions are oriented towards the programming of symmetric multiprocessors (SMP's), as opposed to distributed-memory machines. Section 2.1 introduces the Cilk language through the example of computing Fibonacci numbers. Section 2.2 then 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, and 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. Section 2.8 describes tools for debugging Cilk code, including a description of how to use the Nondeterminator data-race detection software. Finally, Section 2.9 describes the computational model supported by Cilk's scheduler, and Section 2.10 describes Cilk's facilities for measuring the performance.

Additional material on multithreaded programming using a Cilk-like model can be found in the minicourse notes [23]. This minicourse teaches 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 a given platform. Cilk's runtime system takes care of details like load balancing and communication protocols. Unlike other multithreaded languages, however, Cilk is algorithmic in that the runtime system's scheduler guarantees provably efficient and predictable performance.

The basic Cilk language is extremely simple. It consists of C with the addition of three 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

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

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

        x = fib (n-1);
        y = fib (n-2);

        return (x+y);
    }
}

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

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

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

(a)

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

cilk int fib (int n)
{
    if (n<2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}

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

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

    sync;

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

(b)

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

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 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

[1] This program is a terrible way to compute Fibonacci numbers, since it runs in exponential time when logarithmic-time methods are known [12, page 850].

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

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

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

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 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,

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

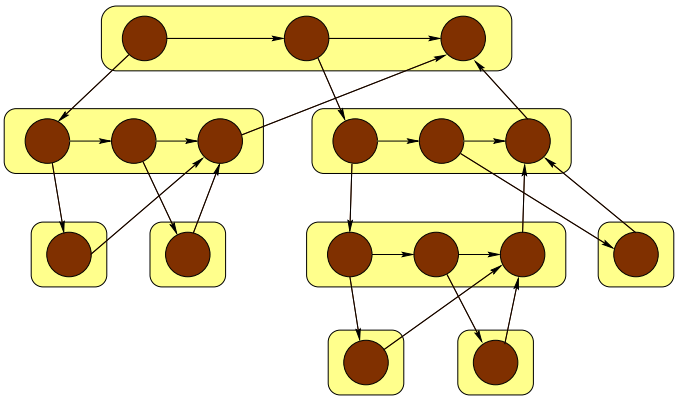

Figure 2.2: The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates 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.

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 2.9.

2.2 Compiling and running Cilk programs

Depending on your target machine, the steps involved in compiling and running Cilk programs may be different. The following instructions are specific to a Sun Microsystems Solaris SMP. We assume that you have basic knowledge of Unix and C program development.

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. The easiest way for a new Cilk programmer to get started is to use the standard `Makefile` included in the `examples` directory of the Cilk distribution, which automates the compilation process.

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,[3] you can simply type:

```
> make fib
```

to make the executable. To run the program, type

[3]If you are using a system on which the Cilk distribution is already installed, it is only necessary to make a copy of `fib.cilk` and `Makefile` from the `examples` directory. Make sure that your copy of the `Makefile` contains a `CILK_DIR` assignment that points to the distribution directory.

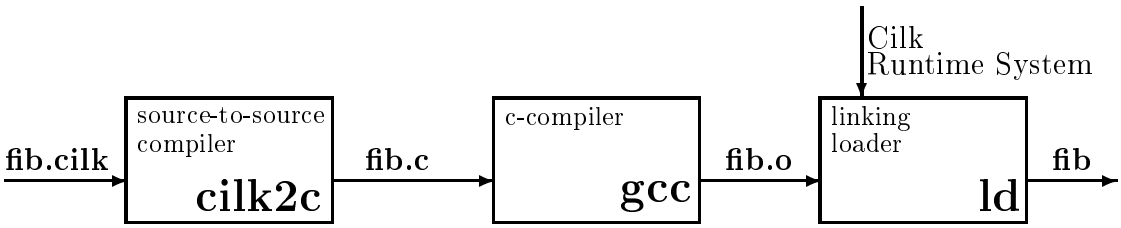

Figure 2.3: Compiling the Fibonacci program. Cilk code is passed through the `cilk2c` compiler, a type-checking preprocessor which generates C output. This output is then compiled with `gcc` and linked with the Cilk runtime library.

```
> 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
```

A Cilk program can be instructed to print more statistics by means of the `-stats` option. The command line

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

yields the following output:

```
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.[4] The meanings of work and critical path will be described in Section 2.9. 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 a application on many few processor than the parallelism, then you should expect linear speedup. Additional statistics can be obtained by setting the statistics level higher than 1.

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

[4] 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.

`-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. Increasing the level prints more and more detailed 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 the runtime system and the application were compiled with. The highest meaningful statistics level is currently 6.

`-no-stats` Equivalent to `-stats 0`, that is, prints neither performance numbers nor statistics.

`-heap` *size* Set a maximum heap of *size* megabytes for use by the library function `Cilk_malloc`.

`-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.

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`.

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. The one exception to this rule is if you use the library version of Cilk (see Section 2.7.4) which allows you to start the Cilk scheduler manually from within a C program and cause it to execute a Cilk subroutine.

The C library was not designed for Cilk, and thus many library functions cause errors if called from a Cilk program. Those C library functions that do not work under Cilk are called ***Cilk-Unsafe***, and those that do work are called ***Cilk-Safe***. The exact set of Cilk-safe functions is platform dependent. In general, functions with static state and file I/O functions are Cilk-Unsafe. For a more detailed discussion of this topic, see Section 6.14.1.

A subset of Cilk-Unsafe library functions, including `rand()` and `malloc()`, have been reimplemented for Cilk. You can use the Cilk version of these functions in place of the standard C version by putting

```
#include <cilk-lib.h>
#include <cilk-compat.h>
```

at the beginning of your source file, in addition to the mandatory inclusion of `cilk.h`. Including the file `cilk-lib.h` provides the Cilk versions, whose names are preceded by `Cilk_`, as in `Cilk_malloc()`. Also, `cilk-lib.h` provides the default C library routines with the prefix `C_`, as in `C_malloc()`. Including the file `cilk-compat.h` causes the standard names of Cilk-Unsafe library functions to be `#define`'ed to the Cilk version. See Section 6.14.1 for details.

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 `Cilk_malloc()` library function and deallocated with `Cilk_free()`. To specify Cilk's version of these functions using their conventional names—`alloca()`, `malloc()`, and `free()`—just put

```
#include <cilk-lib.h>
#include <cilk-compat.h>
```

at the beginning of your source file, in addition to the mandatory inclusion of `cilk.h`. In the remainder of this section, we shall assume that the compatibility library header file is included and so that Cilk's version of these functions is used.

2.4.1 Stack memory

Cilk uses a ***cactus stack*** [25] for stack-allocated storage, such as is needed for procedure-local variables. As is shown 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 [25]. 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

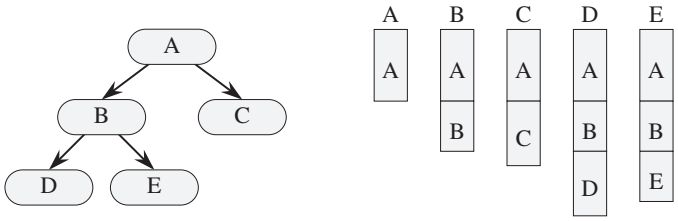

Figure 2.4: A cactus stack. Procedure A spawns B and C, and B spawns D and E. The left part of the figure shows the spawn tree, and the right part of the figure shows the view of the stack by the five procedures. (The stack grows downward.)

returns. Similarly, sibling procedures cannot reference each other's local variables. Just as with the C stack, pointers to objects allocated on the cactus stack can only be safely passed to procedures below the allocation point in the call tree.

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

```
ptr = alloca(size);
```

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

In the current release, Cilk's version of `alloca()` does not work properly when it is called from within a C function. This problem will not be remedied soon. See Section 6.14.1.

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 `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

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

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

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

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

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

cilk void bar (int *px)
{
  *px = *px + 1;
  return;
}
```

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

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.

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`.[5] 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 Nondeterminator described in Section 2.8 is a debugging tool that can detect automatically whether a Cilk program run on a given input has a race.

The easiest way to deal with the anomalies of shared access is simply to avoid

[5]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.

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 `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` to be 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()` (see Section 4.3)

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`.[6] Acquiring and releasing a Cilk lock has the memory consistency semantics of release consistency [18, 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:

```
Cilk_lockvar mylock;
      :
      :
{
   Cilk_lock_init(mylock);
      :
   Cilk_lock(mylock); /* begin critical section */
      :
      :
   Cilk_unlock(mylock); /* end critical section */
}
```

A simple example of code that uses locks is the bucket-sort example in Section 3. A more complicated example can be found in the Barnes-Hut code in Section 3.

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

[6] 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.

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);
```

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 [13].

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.

```
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.

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 that 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.*

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 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

```
#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;
}

  /* 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`.

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.[7]

By testing `SYNCHED`, you can sometimes conserve space resources and obtain better locality than 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 [24] 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

[7] 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.

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.

2.7.4 Library version

The library version of Cilk allows otherwise serial programs written in C, C++, or another language to call a Cilk function to perform some parallel task. It also aids in interfacing with various libraries such as windowing libraries. The library version of Cilk is based on the threaded version of Cilk.[8]

The first step a program must take to use the Cilk library version is to initialize the library. The function `Cilk_init` takes the given set of Cilk options and prepares the run time system to run Cilk functions. These options include many processors should be used (the normal `-nproc` argument). These options can either be filled then explicitly, or they can be parsed out of the command line by the `Cilk_parse_command_line` function. `Cilk_parse_command_line` first initializes the `Cilk_options` structure passed in to its default value. It then parses any Cilk options at the beginning of the command line into the `Cilk_options` argument and removes them by modifying the argc and argv passed in.

Once the library has been initialized, any number of Cilk function can be run using the `Cilk_run` function. The `Cilk_run` function takes a pointer to a Cilk function and a void pointer argument, runs the Cilk function on the given argument, and finally

[8]See Appendix A for details on configuring the library version.

returns the integer result of the Cilk function. sense Cilk run is intended to used in a serial program, it is not reentrant. So, it may not be used from within a Cilk function, and if the program using Cilk run is multi-threaded, the programmer must make certain that no `Cilk_run` calls overlap.

Finally, before the program terminates it should call the `Cilk_finish` function to shut down all of Cilk's worker threads.

The Cilk distribution includes a simple example of the Cilk library version in the `examples/library` directory.

2.8 Debugging

Parallel programs are harder to debug than sequential programs, because much more can be going on at any single moment. In Cilk, the execution of a program can be affected by scheduling, synchronization, and other factors which are not present in a serial C program. Since Cilk semantics are lifted from C, many common bugs can be found by debugging the C elision

Moreover, although many Cilk programs are intended to be deterministic no matter how their threads are scheduled, bugs may nevertheless cause them to behave nondeterministically. This section describes a novel debugging tool, the Nondeterminator, which can greatly simplify the finding of "data race" bugs.

Precisely, a ***data race*** exists when two logically parallel threads access the same memory location while holding no locks in common, and at least one of the threads writes the location. Figure 2.13 shows an example of a data race in a Cilk program. Procedures `foo1`, `foo2`, and `foo3` all access the shared variable `x`. The accesses by `foo1` and `foo2` are "protected" by lock `A` and hence do not form a race. Likewise, the accesses in `foo1` and `foo3` are protected by lock `B` and so don't form a race. The accesses by `foo2` and `foo3` are not protected by a common lock, however, and so form a data race. Because these accesses are not atomic, their machine instructions may interleave, possibly resulting in a lost update to `x`. The final value of `x` printed by `main` may therefore be 2, 3, or 6.

Data races may be intended by the programmer, but they are likely to be bugs. The Nondeterminator debugging tool finds data races by executing Cilk programs on a single processor and examining the generated computation. Because the Nondeterminator executes the program, it only finds data races that occur for the particular input given to the program. The Nondeterminator is guaranteed, however, to find all data races in the computation generated by running the program serially on that input.

The easiest way to use the Nondeterminator is through the standard `Makefile` included in the `examples` directory of the Cilk distribution. The source code of the example from Figure 2.13 can be found under the name `data-race.cilk` in the `examples/nd` directory of the Cilk distribution. After installing the distribution, you can simply type:

```
> make data-race.nd
```

```
int x;
Cilk_lockvar A, B;

cilk void foo1() {
  Cilk_lock(A);
  Cilk_lock(B);
  x += 5;
  Cilk_unlock(B);
  Cilk_unlock(A);
}

cilk void foo2() {
  Cilk_lock(A);
  x -= 3;
  Cilk_unlock(A);
}
```

```
cilk void foo3() {
  Cilk_lock(B);
  x++;
  Cilk_unlock(B);
}

cilk int main() {
  Cilk_lock_init(A);
  Cilk_lock_init(B);
  x = 0;
  spawn foo1();
  spawn foo2();
  spawn foo3();
  sync;
  printf("%d", x);
}
```

Figure 2.13: A Cilk program with a data race.

to make the executable linked with the Nondeterminator's runtime system. To run the program, type

```
> data-race.nd
```

After the program runs, you should see a printout similar to the following:

```
---------------------------------------
Data race @ 0x2daa8
   `x' (data-race.cilk:17) with
   `x' (data-race.cilk:23)
---------------------------------------
```

along with other output from the Cilk program and runtime system. The Nondeterminator finds a race in the program as we expect. Besides the memory address involved in the race, the variable names of the two competing accesses are shown, along with their line numbers and file names.

The Nondeterminator also correctly detects races involving Cilk's memory allocation routines. This may result in a slightly different appearance of race reports; see Section 9.2.

In addition to user locks, Cilk provides atomicity implicitly through the use of inlets. The Nondeterminator treats inlets as if they were regions of code surrounded by the appropriate lock. In the case of implicit inlets, the Nondeterminator can detect whether the inlets commute, and will report as a race two implicit inlets that do not commute. For full information on the Nondeterminator's handling of inlets, see Section 9.7.

The Nondeterminator reports each race only once. That is, if the Nondeterminator has already reported a race between two pieces of text in the code, it will not report a race between those two again. (It may, however, report multiple races on the same memory location.) Sometimes, you may only want to see the report of the first race in the entire program. You can use the runtime option `-first-stop` to let the program terminate whenever the first race is found.

Like other debugging tools, the Nondeterminator runs in time proportional to the ordinary serial execution of the program. For codes that use locking heavily, the Nondeterminator may seem to run too slowly. In that case, you may want to try running with the `-brelly` option. This causes the Nondeterminator to run a different, faster algorithm that detects violations of the "umbrella locking discipline" rather than data races. The ***umbrella locking discipline*** requires that each memory location be protected by the same lock within a parallel subcomputation. Threads that are in series may use different locks for the same location (or possibly even none, if no parallel accesses occur), but if two threads in series are both in parallel with a third and all access the same location, then all three threads must agree on a single lock for that location. With the `-brelly` option, the Nondeterminator still detects data races, because every data race is an umbrella discipline violation, but the Nondeterminator may also detect some violations that do not actually constitute data races. You will need to examine each violation to determine if it is a data race or not. For more details, see Section 9.3.

The Nondeterminator can find races only involving static (global) variables or memory allocated by `Cilk_malloc`, and only in code that is compiled with `cilk2c`. The `cilk2c` compiler issues warnings when a Cilk program calls a function (or library) in a different file, as there may be races involving that function. Furthermore, the Nondeterminator may report "false races" when the program uses locks to implicitly synchronize control. Each case where the reports of the Nondeterminator may not be precisely accurate is detailed in Chapter 9, along with suggested techniques for dealing with those problems.

2.9 The Cilk model of multithreaded computation

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

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 [7]. Fortunately, every Cilk program generates a well-structured dag that can be scheduled efficiently [8].

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

Cilk's work-stealing scheduler executes any Cilk computation in nearly optimal time. From an abstract theoretical perspective, there are two fundamental limits as to how fast a Cilk program can run. Let us denote by T_P the execution time of a given computation on P processors. The ***work*** of the computation is the total time needed to execute all threads in the dag. We can denote the work by T_1, since the work is essentially the execution time of the computation on one processor. Notice that with T_1 work and P processors, the lower bound $T_P \geq T_1/P$ must hold.[9] 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 [6], and the formula

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

is a good approximation of runtime. This performance model assumes an ideal parallel computer, that locks are held for constant time, and that locking is statistically random (i.e., no hot spots). "Your mileage may vary."

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 $P \ll \overline{P}$, meaning that 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 the Cilk program

[9] This abstract model of execution time ignores memory-hierarchy effects, but is nonetheless quite accurate [6].

is guaranteed to run with almost perfect linear speedup. The measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs over the entire range of possible parallel machine sizes. Cilk's timing instrumentation (see Section 6.13.2) can calculate these two measures during a run of a program, no matter how many processors are used to run the program.

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 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 [4]), but the bound $S_P \leq S_1 P$ serves as a reasonable limit. If a computation uses moderate amounts of memory when on one processor, you can be assured that it will use no more space per processor 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.

2.10 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 2.14. 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. Due to the "Heisenberg uncertainty" effect caused by a measurement perturbing what is being measured, it is important to understand how to generate the measurements you wish.

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* with the following definition in the associated application `Makefile`:

   ```
   CILK_CRITICAL_PATH = 0
   ```

2. *Run the program* without bothering about the runtime option `-stats` (see Section 2.2). The statistics gathered are total runtime and work only. Different levels of the `-stats` option do not provide significant insight.

	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 2.14: The performance of example Cilk programs. Times are 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.

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. Cilk is used for performance analysis as follows:

1. *Compile the application* with the following definition in the associated application Makefile:

   ```
   CILK_CRITICAL_PATH = 1
   ```

   This definition enables the critical path measurement within the application. If `CILK_CRITICAL_PATH` is set, a runtime penalty may be paid, depending on the application and the system you are running on.

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

Figure 2.14 shows speedup measurements that were taken of the example programs described in detail in Chapter 3. We measured execution time one one processor T_1, 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.

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`. All measurements in Figure 2.14 were taken with the option `-benchmark`.

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 Cilk reports work and critical-path length measurements, it enables meaningful speedup measurements of programs whose work depends on the actual runtime schedule. Conventionally, speedup is calculated as the one-processor execution time divided by the parallel execution time. This methodology, while correct for deterministic programs, can lead to misleading results for nondeterministic programs, since two runs of the same program can actually be different computations. Cilk's instrumentation can compute the work on any number of processors by adding up the execution times of individual threads, thereby allowing speedup to be calculated properly for nondeterministic programs.

As can be seen from the table, all of the programs exhibit good speedups. Even the complicated and irregular `Barnes-Hut` code achieves a speedup of 7.6 on 8 processors [27].

The right portion of Figure 2.14 compares execution times where the critical-path length measurement 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

Example Programs

The Cilk distribution contains a variety of example programs which explore the difficulty of solving problems in parallel. Some of these programs, such as the program for computing a sparse Cholesky factorization, have irregular inputs. Others, like the backtrack searching algorithm used to solve the n-queens problem, have an irregular structure in the computation. Because of Cilk's flexibility in expressing parallelism, irregular problems pose no undue hardship on execution efficiency. The minimal loss of performance that is sometimes experienced is generally due to parallel algorithms that are intrinsically less efficient than the serial algorithm they replace. Execution times of the example programs are reported in Section 2.10.

The example programs can be found in the `examples` directory of the Cilk distribution. By working through the source code you can get an idea of how parallelism can be expressed in Cilk. Common solutions to some of the finer points of parallel programming in Cilk can be learned from studying the source code. Please send questions about these programs to `cilk-support@supertech.lcs.mit.edu`.

The next few sections give a short description of each example program. Example programs can be compiled by typing `make <program-name>`. The `Makefile` in the example directory can be used as a template for your own programs. When copying `Makefile` to a different location, be sure to change the `CILK_DIR = ..` line at the top to point to the proper location of the Cilk distribution.

Upon execution each of the example programs reports execution time, work, critical path and parallelism. In general the example programs measure both overall performance and the core portion of the algorithms. To get more detailed statistics about overall performance use the Cilk option `-stats <n>`. Other useful Cilk options include `-debug <n>` to print out debug information and `-help` to list the Cilk options that the runtime system accepts. If the program did not print out critical path and parallelism, then the line `CILK_CRITICAL_PATH = 0` in the makefile should be changed to `CILK_CRITICAL_PATH = 1`. After `touch`ing the `.cilk` file, and remaking the program, the numbers should appear. Turning this option on typically slows down the program; hence, it should only be used in the design and debug phases. Note that due to scheduling differences and related cache effects the times for critical path and parallelism may differ from run to run, especially if the number of processors is also

Overview

`barnes-hut`	Barnes-Hut n-body calculation, written by Keith Randall.	p. 32
`blockedmul`	Blocked multiplication of two dense $n \times n$ matrices, written by Keith Randall.	p. 40
`bucket`	Bucket sort of a random set of floating-point numbers, written by Guang-Ien Cheng.	p. 32
`cholesky`	Cholesky factorization of a sparse symmetric positive-definite matrix represented as a quad-tree, written by Aske Plaat and Keith Randall.	p. 33
`cilksort`	Sort a random permutation of n 32-bit integers, written by Matteo Frigo and Andrew Stark.	p. 34
`ck`	Noninteractive Checkers program, written by Don Dailey.	p. 35
`fft`	Fast Fourier transformation of a vector of length n, written by Matteo Frigo.	p. 35
`fib`	Simple Cilk example to compute Fibonacci numbers.	p. 35
`game`	Find a winning strategy for a simple game. Tests `abort`, written by Matteo Frigo.	p. 36
`heat`	Heat-diffusion calculation on an $m \times n$ mesh, written by Volker Strumpen.	p. 36
`kalah`	Interactive Kalah game, written by Reid Barton.	p. 37
`knapsack`	Solve the 0-1 knapsack problem on n items using branch and bound, written by Matteo Frigo.	p. 37
`knary`	Synthetic benchmark to generate arbitrary sizes of serial and parallel work, written by Robert D. Blumofe.	p. 38
`lu`	LU-decomposition (without pivoting) of a dense $n \times n$ matrix, written by Robert D. Blumofe.	p. 39
`magic`	Compute the number of magic squares of size $n \times n$, written by Matteo Frigo.	p. 40
`notempmul`	A slightly less parallel, but more efficient, blocked multiplication of two dense $n \times n$ matrices, written by Keith Randall.	p. 40
`plu`	LU-decomposition with partial pivoting, written by Svetoslav Tzvetkov.	p. 39
`queens`	Backtrack search to solve the problem of placing n queens on an $n \times n$ chess board so that no two queens attack each other, written by Keith Randall.	p. 41
`spacemul`	Another matrix multiplication algorithm, written by Keith Randall.	p. 40
`rectmul`	Matrix multiplication of nonsquare matrices, written by Harald Prokop.	p. 40

changed.

If an error message appears that memory is exhausted, try specifying a larger heap for Cilk, through the `-heap <size>` option. `-heap 500` sets the maximum size the heap can grow to, to 500 Megabytes.

The example programs accept the option `-benchmark [short|long]` in addition the described command lines.

Barnes-Hut N-Body Code

The executable for the Barnes-Hut code is the file `code` in the `Barnes-Hut` subdirectory. It can be built by typing `make` in the same subdirectory.

Algorithm Description

The Barnes-Hut algorithm simulates the interaction of n point masses under the interaction of gravity. This simulation is used in astrophysics to explore the dynamics of galaxies and galaxy formation. A naive algorithm to compute gravitational interactions would run in $\Theta(n^2)$ time, computing an interaction for each pair of masses. In the Barnes-Hut algorithm, many distant masses are approximated by a single large mass in order to reduce the number of force calculations required. An oct-tree is used to construct these approximations and determine when such approximations can be used without introducing too much error. The running time of the Barnes-Hut algorithm, assuming a balanced oct-tree, is $\Theta(n \lg n)$. In the Cilk code, the four phases of the computation are parallelized. To run the program for two force calculations on 64K particles call `code -nproc` n `< sample-input`

Remarks

This Cilk Barnes-Hut code is derived from a serial C code which can be obtained from Joshua Barnes's home page (`http://www.ifa.hawaii.edu/~barnes/index.html`).

Bucket Sort

Command

`bucket [-n #] [-benchmark] [-h]`

Algorithm Description

Bucket sort is a linear time algorithm for sorting a set of n numbers evenly distributed over a fixed range. The current implementation is primarily meant as a simple example of using Cilk's locking features and is not optimized for performance. It also contains a rudimentary parallel implementation of a prefix-sum computation on an array, which is used as a subroutine in the sort.

The program sorts a random array of n floating-point elements in the range $[0, 1)$. The algorithm operates by creating n buckets, each of which contains an initially-empty list of elements. The n elements are distributed into the n buckets based on a simple mapping function: an element x goes to the $\lfloor xn \rfloor$'th bucket. The elements in each bucket are kept in sorted order through insertion, so that after all elements have been distributed to buckets, the sorted array can be recovered by going through the buckets in order and listing the elements in each. As the expected number of elements per bucket is constant, the expected work of the algorithm is linear.

Since the distribution of elements into buckets is performed in parallel, the program includes a `Cilk_lockvar` field in each bucket and uses the `Cilk_lock` and `Cilk_unlock` functions to ensure that only one thread is inserting into each bucket at a time. After all elements have been distributed, the code computes in parallel the prefix-sum of the the number of elements in each bucket to find out how many elements come before each bucket in the sorted order. This information is then used to place the bucket elements back into the original array in sorted order.

Sparse Cholesky Factorization

Command

`cholesky [-n` *size*`] [-z` *nonzeros*`] [-f` *filename*`] [-benchmark] [-h]`

Algorithm Description

This program performs a divide and conquer Cholesky factorization of a sparse symmetric positive definite matrix ($A = LL^T$). Using the fact that the matrix is symmetric, Cholesky does half the number of operations of LU. The method used is the same as with LU, with work $\Theta(n^3)$ and critical path $\Theta(n \lg n)$ for the dense case. A quad-tree is used to store the nonzero entries of the sparse matrix. Actual work and critical path are influenced by the sparsity pattern of the matrix. After the factorization `cholesky` multiplies the factors and checks whether the factorization was correct. The difference is printed and should be zero.

The input matrix is either read from the provided file or generated randomly with size and nonzero-elements as specified. Run with `cholesky -f george-liu.mtx` for a given matrix, or `cholesky -n 1000 -z 10000` for a 1000×1000 random matrix with 10000 nonzero elements.

Remarks

Matrices can be found on the web on a site called Matrix Market (`http://math.nist.gov/MatrixMarket/`). The sparse Cholesky example understands the Matrix Market data format. In many sparse Matrix Market matrices the actual floating-point numbers are not included. `cholesky` generates random val-

ues for the nonzeros automatically. A tiny test matrix, showing the input format, is provided as `george-liu.mtx`.

`cholesky` consists of many recursive calls in order to traverse its quad-tree data structure, causing a 20–25 percent Cilk overhead.

This code combines the symbolic and numerical factorization phases, unlike some other sparse codes. At the leaves of the quad-tree a blocked algorithm is performed to reduce the tree traversal overhead, at the cost of storing some zero entries.

This sparse Cholesky code is not as sophisticated as other codes, such as the one found in the Splash-2 suite [30]. State-of-the-art codes restructure the matrix to perform the actual factorization with dense parallel linear algebra routines that do not have the tree-traversal overhead. The code makes for a good example, however, because its simple divide-and-conquer structure is comparatively easy to understand, and its efficiency is more portable. In contrast, codes, such as that in the Splash-2 suite, must be re-tuned for efficiency whenever it is ported to a machine with caches of different sizes. It may be possible to combine the advantages of state-of-the-art codes with the parallel divide-and-conquer strategy used by this sparse Cholesky code, but we have not yet pursued this research direction.

Cilksort

Command

```
cilksort [-n #] [-benchmark] [h]
```

Algorithm Description

Cilksort is a parallel sorting algorithm, donned "Multisort", which is a variant of ordinary mergesort. Multisort is inspired by [1]. Multisort begins by dividing an array of elements in half and sorting each half. It then merges the two sorted halves back together, but in a divide-and-conquer approach rather than the usual serial merge. Say that we wish to merge sorted arrays A and B, with A being the larger. We begin by dividing array A in half, letting A_1 denote the lower half and A_2 the upper. We then take the middle element of A and use a binary search to discover where that element should fit into array B. This yields a division of array B into subarrays B_1 and B_2. We then recursively merge A_1 with B_1 and A_2 with B_2, which yields the desired fully merged version of A and B. This algorithm has work $\Theta(n \lg n)$ and critical path $\Theta(\lg^3 n)$.

The implementation falls back on a serial quicksort and mergesort for small arrays. The code includes an ordinary serial quicksort with some well-known optimizations [28].

Remarks

The quicksort implementation on its own takes 0.95 seconds to sort 1,000,000 32 bit numbers on a single UltraSparc processor. The Multisort implementation on the same input requires 1.15 seconds on one processor (an overhead of 21%), but is improved to 0.15 seconds on 12 processors, a 10 times speedup. Furthermore, measurements (on 12 processors) show a parallelism of 300, suggesting that the implementation will continue to scale to larger numbers of processors.

Checkers

Command

`ck [-w #] [-b #] [-d #] [-benchmark] [-h]`

Algorithm Description

This program plays rudimentary checkers, without double jumps. It is not interactive, both colors are played by the computer. Also a good test for `aborts`. The option `-d #` specifies how deep the game tree is evaluated. To use different search depths for black and white use `-b #` and `-w #`. The following options have black do a 5 ply search and white 9 ply: `ck -w 9 -b 5`.

Fast Fourier Transform

Command

`fft [-n #] [-c] [-benchmark] [-h]`

Algorithm Description

This program is a highly optimized version of the classical Cooley-Tukey Fast Fourier Transform algorithm [11]. Some documentation can be found in the source code. The program is optimized for an exact power of 2. To test for correctness use parameter `-c`.

Use the `-heap size` runtime option for sizes larger than 1024, where a good choice for `size` is the amount (in Megabytes) of main memory that your system has.

Fibonacci

Command

`fib [-n #] [-benchmark] [-h]`

Algorithm Description

This program computes the nth Fibonacci number [15] in the probably least efficient way. It does, however, allow a parallelization in Cilk that is as close to ordinary C as you can possibly get. All it takes is plugging in the `spawn` and `sync` keywords in the logical places. `fib` should give an almost perfect speedup, if reasonably large Fibonacci numbers are computed. The file `fib-serial.c` contains the serial version of `fib`, for comparison.

Game

Command

```
game [-h]
```

Algorithm Description

This is a simple game to test `abort`s. It is not interactive. Rules: There are two players on a 5×5 board. Alternatively, each player puts a coin in position (i, j) and in all positions (i', j') for all $i' \leq i$ and $j' \leq j$ (multiple coins count as one). The player who fills the board loses. The first player has a winning strategy.
This game was suggested by David Finberg, `dfinberg@mit.edu`.

Heat Diffusion

Command

```
heat [-g #] [-nx #] [-ny #] [-nt #] [-benchmark] [-h]
```

Algorithm Description

This program uses a Jacobi-type iteration to solve a finite-difference approximation of parabolic partial differential equations that model for example the heat diffusion problem. The iteration formula for a two-dimensional domain $\{i, j\}$, based on a five-point-stencil, is as follows:

$$u_{i,j}^{(k+1)} = c_1 u_{i-1,j}^{(k)} + c_1 u_{i+1,j}^{(k)} + c_2 u_{i,j}^{(k)} + c_3 u_{i,j-1}^{(k)} + c_3 u_{i,j+1}^{(k)},$$

where u may be interpreted as heat value, c_1, c_2 and c_3 are constants, and i, j and k span an index domain. The iteration is performed in the k-direction. Parallelization can be achieved by partitioning the $\{i, j\}$-domain.

`heat` executes `nt` steps of the heat diffusion simulation on a grid with `nx` columns and `ny` rows. The parameter `g` specifies the granularity of the computation. Given the total number of columns the number of partitions is `nx/g`. Each partition is computed by a single Cilk thread. Due to data dependencies, threads must be `spawn`'ed and `sync`'ed within a single iteration, enforcing a global synchronization per iteration.

Kalah

Command

`kalah [-depth #] [-hash] [-nohash] [-hashsize #] [-stones #]`

Rules

Kalah is played by two players on a board with two rows of 6 holes facing each other and two "kalahs" (the rightmost hole of each player). At the beginning there are 6 stones in every hole and the kalahs are empty. You can specify the number of stones initally placed in the holes using the `-stones` option.

The object of Kalah is to get as much stones into your own kalah by distributing them. A player moves by taking all stones of one of his 6 holes and distributing them counterclockwise by putting one stone in every hole including his but excluding the opponent's kalah.

If the last stone is distributed into his own kalah, the player may move again. He has to move again even if he does not want to. If the last stone falls into an empty hole of the player and the opponent's hole above (or below) it is not empty the player puts his last stone and all the stones in his opponent's hole into his kalah. He has won all those stones.

The game ends if all 6 holes of one player become empty (no matter if it is this player's move or not). The stones in the other player's holes are given into this player's kalah.

The player who won more stones (at least 37) becomes the winner.

Algorithm Description

This program is interactive. The computer searches the game tree `-depth` ply deep. The default is 10 ply.

Knapsack

Command

`knapsack [-f` *filename*`] [-benchmark] [-h]`

Algorithm Description

The 0-1-Knapsack is a standard combinatorial optimization problem: "A thief robbing a store finds n items; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take?" [12, p. 335]

As in queens, the algorithm uses speculative parallelism, but unlike queens, it does not use `abort`. The program reads the list of items with weight and value from a specified problem file. Three example files are provided. The program keeps track of the current best filling in a global bound, shared by all processors in hardware shared memory. This is where speculativity comes in: due to scheduling differences the bound may be lowered faster in some runs than in others.

This program depends on a stronger consistency model than dag consistency. It violates one of the portability recommendations elsewhere in this manual.

Remarks

Comparing runs for different input files provides more insight into the reasons for super/sublinear speedup and speculative parallelism. The superlinear speedup is due to the global "best-so-far" bound, which prevents branches from being searched if it can be determined that, no matter how remaining items are packed, a better solution will not be found.

With a random input with only small variations in value/weight ratio, parallelism is not defeated by heuristic optimizations since this optimization doesn't kick in until late in the search, when many branches have been spawned, and thus work is saved in parallel runs compared to the serial run, since there is more chance of finding the best bound in the parallel run. With large variations in the value/weight ratio, however, branches are pruned from the very beginning, leading to a largely serial run, no matter how many processors are used, giving bad speedup figures.

Knary

Command

`knary [-s` *serial*`] [-p` *parallel*`] [-i` *iter*`] [-d` *depth*`] [-benchmark] [-h]`

Algorithm Description

Knary is a synthetic benchmark. Its name really is k-n-ary. It is a synthetic benchmark whose parameters can be set to produce a variety of values for work and critical path length. It generates a tree of depth *depth*. At every level it spawns *serial* children serially and then *parallel* in parallel. At each node of the tree, the program runs an empty "for" loop for *iter* iterations. This program was written by Bobby Blumofe. For more on Knary and its relation to the dag it generates, see his thesis [3] (available from the Cilk homepage). Playing around with different numbers for the four arguments can be instructive, since it gives insight in how scheduling is influenced by the relative sizes of the blocks of work.

Dense LU Factorization

Command

`lu [-n` *size*`] [-c] [-o] [-benchmark] [-h]`

Algorithm Description

LU performs the divide and conquer form of a blocked LU decomposition of a dense $n \times n$ matrix ($A = LU$), where n is a power of 2 and greater than 16. LU factorization is the most time consuming step of a common method of solving a system of linear equations, finding $\bar{x}$ in $A\bar{x} = \bar{b}$, where A is a matrix and $\bar{x}$ and $\bar{b}$ are vectors. The work of LU is $\Theta(n^3)$, and the critical path is $\Theta(n \lg n)$ [4].

Note that usually LU decomposition is expressed as a triply nested for-loop. Cilk is more suited for a divide and conquer strategy (giving the added benefit of locality), hence our choice of the divide and conquer version of LU. On the downside, the recursive divide and conquer algorithm is not the easiest of codes to comprehend.

Remarks

This version does not do pivoting, and therefore, it should only be used on matrices for which numerical stability is not an issue. An analysis of this algorithm can be found in [4]. This program only factors the matrix. It does not perform the subsequent forward and backward substitution phase needed to solve $L\bar{y} = \bar{b}$ and $U\bar{x} = \bar{y}$.

Dense LU Factorization with Partial Pivoting

Command

`plu [-n` *size*`] [-benchmark] [-h]`

Algorithm Description

The program performs a LU decomposition using partial pivoting on a dense square $n \times n$ matrix ($PA = LU$). n is usually a power of 2, otherwise the matrix is padded with 0's and the identity matrix to the next power of 2.

The algorithm is based on a divide and conquer approach that tries to maximize the locality of reference. Basically, this is a parallelized version of the algorithm described in [29]. The work of the algorithm is $\Theta(n^3)$ and the critical path is $\Theta(n^2)$. The large critical path is due to the operations needed to perform pivoting. Pivoting allows the factorization to work on any nonsingular matrix (which is the major improvement from `lu.cilk`). (See also [29].)

Remarks

The memory organization used for the storage of the matrix is block row major order. The block structure increases the benefit of locality. However, the relatively complicated address arithmetics, makes the functions that deal with the base cases somewhat harder to comprehand.

Magic Squares

Command

`magic [-n #] [-benchmark] [-h]`

Algorithm Description

The program computes the number of magic squares of size $n \times n$, in an original way. It generates a program for a nondeterministic virtual machine and executes it.

In a magic square the sum along all horizontal, vertical, and diagonal lines is the same. For example this is a magic square:

8	1	6
3	5	7
4	9	2

Matrix Multiplication

Command

Matrix multiplication comes in five versions:

```
blockedmul [-n #] [-c] [-benchmark] [h]
notempmul  [-n #] [-c] [-benchmark] [h]
spacemul   [-n #] [-c] [-benchmark] [h]
rectmul    [-x #] [-y #] [-z #] [-c] [-benchmark] [h]
```

Algorithm Description

The program `blockedmul` is an $\Theta(n^3)$-work, $\Theta(\lg^2 n)$-critical-path, square matrix multiplication algorithm [4]. It uses divide-and-conquer to solve one $n \times n$ multiplication problem by splitting it into 8 $n/2 \times n/2$ multiplication subproblems and combining the results with one $n \times n$ addition. A temporary matrix of size $n \times n$ is allocated at each divide step. A serial C matrix multiplication routine is called to do the base case, a 16×16 multiplication. The program `notempmul` is an $\Theta(n^3)$-work, $\Theta(n)$-critical-path, square matrix multiplication program. At the expense of an asymptotically longer critical path, `notempmul` requires no temporary storage and is slightly faster than `blockedmul` on small numbers of processors. The program `spacemul` uses the

`SYNCHED` keyword to obtain an $\Theta(\lg^2 n)$ critical path, while at the same time using less temporary space. The `SYNCHED` keyword allows the program to avoid allocating a temporary matrix when all sub-multiplies are executed in serial on the same processor. `rectmul` multiplies an $x \times y$ matrix by a $y \times z$ matrix in parallel. Use `-c` to check for correctness.

N-Queens

Command

```
queens [-n #] [-benchmark] [h]
```

Algorithm Description

The objective of this familiar problem is to place n queens on an $n \times n$ chess board in a configuration in which they do not attack each other. The nondeterministic algorithm uses `abort` as soon as it has found a solution to stop the computation. Since `queens` is a nondeterministic program, it does not necessarily perform exactly the same computation as the C code. The serial C code examines these placements one by one. In contrast, the Cilk code, examines several possibilities for the placement of queens in parallel. Moreover, as soon as the Cilk code determines that one of the placements is a solution, it aborts the others. Consequently, an apparent super-linear speedup can occur. If it were known in advance which branch the Cilk code chooses, then in principle, one could modify the C program to follow that branch first, yielding a "better" serial code.

Part II — Implementation Notes

Chapter 4

Cilk Runtime System Implementor's Guide

Cilk 5.2 has been designed to be easily portable to modern versions of Unix that support some kind of shared memory. We shall assume that `gcc` is available on the target system; it is possible to get rid of this requirement, but we do not recommend it.

This chapter is intended to give you an idea of what is involved in porting Cilk, but does not explain all the details. In other words, we say what is important to do, but we cannot tell you *how* to do it. We assume that you are familiar with Unix. If you wish to port Cilk, we encourage you to contact `cilk-support@supertech.lcs.mit.edu`.

In order to port Cilk, you must first understand three concepts: how Cilk views the address space, how Cilk uses locks, and how Cilk interacts with the underlying memory model. Having done that, you can start from an existing port and do the suitable modifications.

4.1 How Cilk 5.2 views memory

A Cilk 5.2 program starts as a single Unix process, which performs some initializations and creates a certain number of workers that in turn do the actual job of executing the program in parallel. Depending on the implementation, the workers can be either other Unix processes, or some kind of threads. In order for the scheduler to work properly, there must be some way for a worker to have both private data and data which are shared with the other workers. Setting up this address space correctly is the most important task in porting Cilk.

Let us call "private" and "shared" segments the regions of memory that contain, respectively, the data private to each worker and the shared data. Every variable in the runtime system is declared with a macro `DECLARE_PRIVATE_VARIABLE` or `DECLARE_SHARED_VARIABLE`, and each use of the variable is wrapped by another pair of macros `USE_PRIVATE` or `USE_SHARED`. These macros can be redefined to adapt to a variety of environments, according to the support provided by the underlying operating

system, compiler and linker.

- The simplest and most elegant case is when there is a working `gcc` and a linker that supports segments (such as the GNU linker on Solaris and Linux/elf).

  `gcc` accepts an `attribute` directive that allows us to specify which segment a certain variable belongs to. Moreover, it is possible to configure the GNU linker to store the initial and final address of a segment into variables. Consequently, we chose the following strategy for our Solaris and Linux ports.

  Each worker is a separate Unix process. By default, Unix processes have only private variables, and thus nothing needs to be done for the private segment. We instructed `gcc` and the linker to store all shared variables in a segment called `Cilk_shared`; at runtime, this segment is mapped in as shared segment, using `mmap()`.

- In absence of compiler and linker support, a possibility is to use Posix Threads. An example is in the Solaris/threads port, that uses Solaris threads but whose concepts are the same. Another example is when workers are created through the `clone()` system call (available on Linux), which can create processes that share the same address space.

  In this case, the address space of threads is shared by default; the difficult thing is to get access to some private variables. Cilk 5.2 is smart enough to be able to work with just *one* private variable, which is then used as a pointer into a structure that contains all the other private variables. The definition of the structure is in turn generated by a `perl` script, and each worker allocates its own structure in different regions of the shared space.

  The trick is how to get the singe private variable. In the Solaris/threads port, we used `gcc`'s ability to store a global variable in a register. Lacking this possibility, it may still be possible to use any mechanism that the underlying thread package provides to manipulate per-thread data. One major problem is that private variables are used in performance-critical sections of code, and thus Cilk will be significantly slowed down if this part is not implemented efficiently.

- A third scenario occurs when all variables are private and there is no way of getting shared variables. This is the case, for example, in a Unix system with no `clone()` and no linker support. It is still possible to run Cilk in this configuration, but its functionality will be somewhat reduced. Contact the Cilk developers for suggestions if you have such a system.

In all cases, Cilk 5.2 needs a way to allocate memory shared among the workers. The allocation is done once and for all by the initial process; after that, Cilk manages the memory itself. In particular, Cilk does not require a shared `malloc`: all the system needs to provide is a big chunk of shared memory (called a *heap* in Cilk's jargon).

A common way to set up the heap is by means of `mmap()`. Examples can be found in the existing ports; notice that the semantics of `mmap()` varies in subtle ways from

system to system: be sure to read the manual before trying to use it (and maybe write a small test program). For example, Solaris allows shared mapping from `/dev/zero` but Linux does not. A sufficiently adventurous user may even try to `mmap()` from files in `/proc` (if it exists).

Another way to set up the heap is by means of `shmget()` and related functions. However, there are usually restrictions in the amount of memory that can be mapped, and therefore this alternative is discouraged.

4.2 Locks in Cilk

Cilk is a parallel program that works in a shared memory environment. A Cilk worker must be able to *lock* certain objects to make sure that no other worker is trying to modify them at the same time.

Basically, any interprocess locking mechanism provided by the operating system will be enough as far as Cilk is concerned (Cilk does not require strong properties of locks, like fairness, etc.). However, there are a couple of issues that should be considered carefully.

1. Cilk assumes that locks carry memory semantics with them. That is, if a worker locks an object, modifies it, unlocks the object, and then the lock is grabbed by another worker, then Cilk assumes that the modifications (and all of them) will be seen by the second worker. If you don't know whether this property is maintained by your architecture, you can probably assume it is. Just keep in mind that this requirement may be a source of mysterious bugs.

2. It is likely that operating system locks will be too expensive. On Solaris, we found that it is much faster to implement locks manually using atomic memory operations (Compare And Swap, Test And Set, Fetch And Add, and the like). On some systems, this may even be the only alternative. Therefore, consider implementing locks in shared memory if you can, but be sure that the requirements of the previous item are fulfilled.

4.3 Interaction with weak memory models

Modern processors do tricks to improve execution speed. For example, a processor can execute instructions in a different order than they are issued. If a program contains a store instruction followed by a load instruction, then the processor may swap the two instructions. The reason is that the result of the load is required to continue execution, and thus the load should be performed as soon as possible. Such optimizations are done in such a way that the sequential behavior of the program does not change; however, the parallel semantics *do* change, and you must know what to do in order to guarantee the correctness of the program.

Most of the time, Cilk protects all its shared data with locks. If locks are implemented properly (as described in Section 4.2) then Cilk will work correctly. However, since locking is expensive, we do not normally lock certain shared variables that are crucial for performance. The program will still execute correctly without locks, provided that you are able to enforce the order of certain memory operations. In order to enforce this order, you must define three functions:

```
static inline void Cilk_membar_StoreStore(void);
static inline void Cilk_membar_StoreLoad(void);
static inline void Cilk_fence(void);
```

`Cilk_membar_StoreStore` enforces the order of writes to memory. Specifically, all stores issued before the membar must be completed before any other write is performed.

`Cilk_membar_StoreLoad` enforces the relative order of loads and stores. Specifically, all stores issued before the membar must be completed before any subsequent load operation is initiated.

`Cilk_fence` makes sure that all memory operations issued before the fence are completed before any memory operations that come after the fence are initiated. `Cilk_fence` subsumes the two `membar` operations, but on some architectures the `membar`s are cheaper than a full memory fence.

On some machines, some (or even all) of these functions may do nothing. For example, Sun's Total Store Order makes the StoreStore memory barrier redundant. Also, if your machine has only one processor, none of these memory functions is needed.

Warning 1 We have tried Cilk with sequential consistency and with TSO, and with no memory models weaker than that. We believe that Cilk will work properly under more relaxed memory models, but this claim is untested. Comments and feedback will be appreciated.

Warning 2 The Linux port assumes sequentially consistent memory. Cilk has been reported to work on a two-processor Pentium. Again, we would appreciate any feedback about this point.

The test program `testall` (in the `examples` directory) should be able to determine whether the memory-barrier functions have been implemented correctly.

4.4 Architecture-specific issues

The existing architecture-specific files provide some other functions that are used by Cilk. Unlike what we discussed so far, porting these functions should be straightforward, or the functionality can be omitted altogether. For example, the code that creates workers in Solaris or Linux is almost identical, and either version will likely

work on other machines. Other things, like the priority control in the Solaris port, are optional (for example, we didn't implement them in Linux). The code for timers is optional (although useful to have), and is also straightforward.

4.5 The Nondeterminator

The Nondeterminator in Cilk 5.2 has been tested on SUN Solaris processes, SUN Solaris threads, SGI Irix threads, and Linux processes. To port the Nondeterminator onto other systems, the following points should be observed:

- When a Cilk program runs, there are two portions of shared memory: a static region and a heap region. The ***static region*** includes all global variables. The ***heap region*** includes all locations obtained from the Cilk memory allocation functions (Section 8.2). The Nondeterminator knows the address range of the heap region from the Cilk memory allocator. The difficulties lie in the address range of the static region because this region is allocated by the linker. If you are using the linker scripts in the Cilk distribution, `_begin_Cilk_shared` and `_end_Cilk_shared` delimit the static region. If you are not using the linker scripts, the static region can be found by using the UNIX system command `nm`. In the Solaris thread and Irix thread port we tested, two linker defined variables `_etext` (or `_fdata`) and `_end` represent the starting and ending address of the static region. For other systems, you should define two macros `STATIC_ARENA_START` and `STATIC_ARENA_END` (in `cilk-nd.h`) as the starting and ending address of the static region.

- The Nondeterminator uses several ***shadow spaces*** for its own internal use. Each shadow space is the same size as the shared memory region. For each location of shared memory, each shadow space has a corresponding location. The Nondeterminator uses the Unix memory mapping primitive `mmap` to fix the starting address of shadow spaces such that any shadow space address can be obtained from the corresponding shared memory address by simple arithmetic or logical operations.

  The default version of the Nondeterminator uses a shadow space, ***lockers***, to store information about memory accesses. The macros defined in `cilk-nd.h`, `STATIC_LOCKLISTS` and `HEAP_LOCKLISTS` tell how to get the *lockers* address from a shared memory address.

  The Brelly version of the Nondeterminator uses four shadow spaces. The ***accessor*** shadow space stores information about accesses to memory. The ***info*** shadow space stores filename and line number information for those accesses, and the ***var*** shadow space stores variable name information. Finally, the ***locks*** shadow space stores information about the locks that were held during those accesses. The macros (defined in `cilk-nd.h`) `STATIC_ACCESSOR`, `HEAP_ACCESSOR`, `STATIC_LOCKS`, and `HEAP_LOCKS`, tell how to get the *accessor* and *locks* shadow

space addresses from a shared memory address. The macros `SHADOW_INFO` and `SHADOW_VAR` take an *accessor* address and return an address in the *info* and *var* shadow spaces, respectively.

- The Nondeterminator allocates its shadow spaces linearly downward. On architectures in which the heap space starts at the high end of the address space, this is simple. On platforms in which the heap begins at the low end of the address space (such as IRIX), we add an offset to all of the shadow space addresses. This offset is set in the macro `SHADOW_SPACE_SHIFT` in `cilk-nd.h`.

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), but we plan to provide a provably good job scheduler in a future release.

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" in to 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 NOP, because all children have completed by each sync when scheduling in this fashion.

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]

[1]We maintain a separate deque instead of using the native C stack to maintain portability.

```
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`.

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 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

Someday, we hope to merge the frame deque and the native C stack, which will further reduce the overhead of the nanoscheduler.

to the procedure's frame. The frame is initialized and pushed onto the deque with 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

```
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`.

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.

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 will be minimal.

5.4 Interactions with weak memory models

In general, the Cilk 5.2 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.

Part III — User's Guide

Chapter 6

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 [17, Appendix B]). As compatibility of the C library with Cilk is a major problem, Section 6.14 discusses some of the issues associated with using the C library in Cilk programs.

The typographic conventions used in this chapter are the same as [17]. In particular:

- Typewriter text (`like this`) denotes Cilk keywords or any other text which should be copied verbatim.
- Italicized text (*like this*) denotes grammatical classes (things like statements and declarations).
- Square brackets (like []) denote optional items.

6.1 Program structure

A Cilk program consists of ordinary ANSI C code plus definitions and declarations of 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.

6.2 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 6.7).

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). 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
```

6.3 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.

6.4 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. In addition, 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 illegal in Cilk.

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

Unlike C, no assignment conversion is done for the assignment operator in a spawn statement, so code like

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

is illegal 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]

[1] We anticipate assignment conversion being added to the semantics of `spawn` for a future release.

6.5 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.

6.6 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:

```
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 a clause of an `if` statement and in other control constructs. A `sync` statement may not appear in an inlet definition.

6.7 Inlet definition

An inlet is a piece of code, normally short, defined in the declaration part of a procedure, to specify the actions to be done when a spawned procedure returns. 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 which 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.

6.8 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 *proc* is then invoked as in the `spawn` statement above. 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`.

6.9 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 useless work in a speculative execution prematurely. Any outstanding children may be stopped before completion, and any return values or side effects of these children may or may not be observed.

6.10 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.

6.11 Shared memory

Variables updated by different procedures are kept coherent according to the rules of dag consistency. (See Section 2.5.) Variables updated by the various threads of a single procedure, including the procedures inlets, are kept coherent according to the rules of sequential consistency, since each thread operates atomically on the variables.

Dynamic stack allocation 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. By including the `cilk-compat.h` header file, the conventional name `alloca` is defined to be `Cilk_alloca`.

[**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. Consequently, C functions should use C's `alloca`, which is available using the compatibility call `C_alloca`. Note that any memory returned by `C_alloca` is private and cannot be seen by other Cilk procedures, or even other threads of the same Cilk procedure. We plan to unify the stack allocator in a future release of Cilk, but the necessary mechanism did not make it into the current release.]

Heap allocation. Heap storage can be allocated using

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

This statement has the same semantics as C `malloc`, but it allows the allocated storage to be shared. By including the header file `cilk-compat.h`, the conventional name `malloc` is defined to be `Cilk_malloc`. It is an error to pass pointers returned by C's `malloc` between different threads, but pointers returned by `Cilk_malloc` are safe to use anywhere.

The following statements for allocating, aligning, and freeing memory have the same semantics as their C cousins:

```
void *Cilk_calloc(size_t nelem, size_t elsize);
void *Cilk_memalign(size_t alignment, size_t size);
void *Cilk_realloc(void *s, size_t size);
void *Cilk_valloc(size_t size);
void Cilk_free(void *s);
```

Including `cilk-compat.h` defines `calloc`, `memalign`, `realloc`, `valloc`, and `free` to use these functions.

6.12 Locks

In a perfect world, you would never need to lock your house or your car. In Cilk, as in real life, however, locks are sometimes required to cope with the inadequacies of the system. In Cilk's case, the synchronization constructs and the dag-consistent memory model are not yet sufficiently strong that mutual-exclusion locks are unnecessary. Thus, we provide locks to the user as a way to complement other features of Cilk.

Cilk locks are declared as type `Cilk_lockvar`. The following statements are available operations on lock variables[2]:

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 [18, 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.

[2]The specification of these functions has been changed from earlier Cilk releases to take a `Cilk_lockvar` object rather than a pointer to a `Cilk_lockvar` object.

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. This type of bug can usually be caught by the Nondeterminator, with a few caveats. See Chapter 9 for details.

6.13 Nonstandard features

Some releases of Cilk contain features that are not supported in all releases. These "nonstandard" features include private global variables, and timers. In addition, an OS-thread version of Cilk can be made available, which provides certain advantages in interfacing with external programs. This section briefly describes these nonstandard features. To see if your installation includes a particular feature, the easiest thing to do is to run a test program.

6.13.1 Shared and private variables

Each global variable declaration is shared by default. That is, all references to a variable in a program refer to the same memory location. This memory location's

value is maintained in a dag-consistent fashion, or in a stronger fashion if the hardware supports it. Sometimes, however, it is useful to declare a variable which is private:

declaration: `private` *type var* ;

Each thread in a Cilk program receives its own logical copy of each private variable. Each private variable is initialized to an undefined value when the thread starts. Once initialized, however, private variables cannot be changed except by the owning thread (unlike shared variables which can be changed by simultaneously executing threads). Private variables are useful for communicating between a Cilk thread and C functions it calls, because these C functions are completely contained in a Cilk thread.

All variables defined in libraries and object files that are not generated using `cilk2c` and not part of the Cilk distribution are `private` by default. The keyword `shared` can be used to override this default and explicitly designate a variable as shared. This default is necessary to eliminate bad behavior exhibited by the C library when its variables are shared. We hope to rectify this ungraceful situation in a future release.

Note: Private variables are currently unavailable when using an OS-thread installation of Cilk (see Section 6.13.3).

6.13.2 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 `CILK_CRITICAL_PATH` set to 1 in your `Makefile`. 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));
```

6.13.3 OS-thread runtime system

Cilk 5.2 can be run on top of lightweight operating-system threads (different from Cilk threads), instead of the heavyweight operating-system processes used in the standard distribution. The OS-thread version is useful for I/O, because the same file descriptors are valid across the entire Cilk program. The OS-thread version also usually works better with library functions (e.g., an X toolkit), because variables that were not passed through `cilk2c` are shared by default. The process version is useful because it allows the user to declare private variables, whereas the OS-thread version does not. To install the OS-thread version, consult the `README` file of the distribution.

The OS-thread version has also been hacked to allow Cilk procedures to be called as library functions. This feature is convenient if a user has a large, existing body of code for which a few subroutines could be parallelized. The interface to allow the subroutines to be parallelized is clumsy, but it seems to work. If you are interested in obtaining this version of Cilk, please email `cilk-support@supertech.lcs.mit.edu`.

6.14 Open issues

This section describes several design issues that have not been fully resolved for Cilk.

6.14.1 C libraries

Libraries pose a big problem for Cilk. Many functions in the standard C library do not work properly with Cilk. Some functions use static data, and crash when they are executed concurrently. In other cases, a function may need data which are meaningful on one processor, but meaningless when a thread is migrated to another processor.

There seems to be no way to predict in advance whether a library function will work with Cilk. Some operating systems (Solaris) characterize C library calls as MT-Safe (safe for multithreading) or MT-Unsafe. While it is likely that an MT-Safe function will work with Cilk, we cannot guarantee proper behavior in all cases. It is unlikely that an MT-Unsafe function will work with Cilk.

The bottom line is that the C library should be used with extreme care, at least until we implement a reasonable subset of `libc` in a Cilk-aware fashion. Moreover, at present, the exact set of Cilk-Safe functions depends on the platform on which Cilk is running.

In order to do anything useful with the system, Cilk provides a memory allocator and a random number generator of its own. The C function `malloc()` and its relatives do not behave as one might expect, because they operate within a process and Cilk operates between processes. For these functions, you should replace your calls to the C library functions with the Cilk versions defined in `cilk-lib.h`. Below is a list of functions that have been written in a Cilk-Safe fashion. The Cilk versions of these functions can be accessed with the `Cilk_` prefix, and the original (probably broken) versions of these functions can be accessed with the `C_` prefix.

```
rand      srand    alloca  malloc  calloc
memalign  realloc  valloc  free
```

To use the Cilk versions of these functions under their conventional names, include the library function header file `cilk-compat.h`.

6.14.2 File and socket I/O

Cilk's I/O capabilities are limited. Currently, Cilk does not have any support for parallel I/O. In particular, file pointers, file descriptors and socket descriptors may not remain valid between threads (that is, across `spawn`'s, `sync`'s, and procedure boundaries). Therefore, for any file or socket I/O in a Cilk application, the file or socket must be opened, operated on, and closed within the same Cilk thread. The use of the OS-thread version of Cilk (see Section 6.13.3) will mitigate, but not eliminate, some of these problems.

In short, Cilk I/O is a research problem that we shall investigate in the future. We plan to add the capability to memory map files and do parallel file and socket I/O to a future release.

Fortunately, buffered I/O from a terminal (`stdin`/`stdout`) appears to work in all the cases we have tried so far.

6.14.3 Unimplemented features

Other unimplemented features include:

- A user-defined clean-up construct called `on_abort` which is invoked when a procedure is aborted prematurely.
- A `superreturn` feature which allows an inlet to force its procedure to return without first resuming the procedure.
- Support for distributed memory multiprocessors.
- Linguistic specification and runtime implementation of stronger memory models.
- An adaptively parallel scheduler which can dynamically grow and shrink a Cilk computation to use whatever processors are available on the parallel computer.

Please let us know what other features you would find useful by sending electronic mail to `cilk-support@supertech.lcs.mit.edu`.

Chapter 7

The Cilk Compiler

Name

`cilk2c` — The Cilk language compiler (type-checking preprocessor).

Synopsis

cilk2c [***options***] [***infile.cilk***]

Description

The type-checking preprocessor `cilk2c` is the compiler for the Cilk language as defined in the *Cilk Language Reference Manual*. It reads from an input Cilk program file, which by convention has the extension `.cilk`, and generates ordinary C code in the output file, which has the same root name as the input file but the extension `.c`. (If the input filename is omitted, `cilk2c` reads from standard input and writes to standard output.) To produce an executable, the generated C file must be compiled using `gcc` and linked with the Cilk runtime library.

The `cilk2c` compiler includes full syntax checking and type checking for Cilk. If your Cilk program contains errors, `cilk2c` will print error messages to standard error, and no C output will be produced.

The following options are available. Multiple options must be specified separately and cannot be stacked.

General options

`-help` Print a help message.

`-options` Print all options.

`-copy` Print copyright information.

`-v` Print version information.

Output options

`-o` ⟨*name*⟩ Write C output to file ⟨*name*⟩

`-N` Omit #line, format for readability

Warning options

`-ansi` Disable GCC extensions and undefine `__GNUC__`

`-W`*n* Set warning level; *n* in 1-5. Default warning level is 4.

`-Wall` Same as -W5.

`-il` Ignore line directives (use actual line numbers).

`-offset` Print offset within the line in warnings/errors.

`-name` *x* Use stdin with *x* as filename in messages.

Macro preprocessor options

`-P`*str* Set the preprocessor command to *str*.

`-pre` Print the preprocessor command and flags.

`-I`*path* Specify path to search for include files.

`-D`*macro*`[=`*Value*`]` Define *macro* (with optional value).

`-U`*macro* Undefine *macro*.

`-H` Print the name of each header file used.

`-undef` Do not predefine nonstandard macros.

`-nostdinc` Do not scan standard include files.

Debugging options

`-lex` Show lexical tokens.

`-yydebug` Track parser stack and actions.

`-insert` Track symbol creation.

`-sym` Print out symbol tables after parse.

`-ast` Print out syntax tree (after last phase).

`-nopre` Don't preprocess.

`-nosem` Don't semantic-check.

`-notrans` Don't transform syntax tree.

`-noprint` Don't write C output.

CC compatibility options

`--` Toggles ignoring of unknown options. Designed for makefile compatibility with your C compiler. For instance, in your makefile you can pass any compiler flags to `cilk2c` by bracketing them with `--`: for instance,

```
cilk2c -- $(CFLAGS) --
```

`cilk2c` will use any options that it recognizes, such as -I and -D, and ignore the rest.

Bugs

The `gcc` compiler is required to compile the C code generated by `cilk2c`, as the C code uses some `gcc` extensions.

There are some known bugs in `cilk2c`'s ANSI C type-checking. The type equivalence test fails to work on array types whose dimensions are not literal constants, so the type-checker fails to report the type disagreement in:

```
extern int a[1+1];
int a[0+1];
```

`cilk2c` cannot detect pointer arithmetic which is nonconstant at load time. For example, it allows the following initialization of the global variable `diff`, which is technically illegal in ANSI C (and rejected by gcc):

```
int i;
int j;
int diff = &j - &i;
```

`cilk2c` does not properly understand the `const` keyword. The following code will fail to compile:

```
const int i = 42;
int a[i];
```

Finally, `cilk2c` fails to detect multiple definitions (i.e., initializations) of an external identifier. For example, `cilk2c` accepts the following top-level definitions without complaint:

```
int x = 5;
int x = 6;
```

Also, `cilk2c` has several bugs and limitations in its Cilk to C translation. A full list of these bugs can be found in the Cilk distribution's `cilk2c` directory, in the file `BUGS`. Our compiler is currently the most buggy of our system's components, a situation which we are actively attempting to remedy.

Internals

See [24] for a description of how `cilk2c` is constructed. If you wish to build a type-checking preprocessor for your own C-based language, the basic C parser and translator are available as the `c2c` program from

`ftp://theory.lcs.mit.edu/pub/c2c/c2c-0-7-3.tar.Z`

Chapter 8

Specification of Cilk Libraries

The Cilk runtime system exports some variables which may be used by Cilk programs. In addition, there are a small number of library functions that Cilk provides, for the user's convenience. The variables are described in Section 8.1, and the library functions in Section 8.2.

In order for Cilk to work at all, every Cilk file must include the header file `cilk.h`, which defines a variety of Cilk internal names. To use the library functions described in this chapter, a program must also include `cilk-lib.h`. The library functions are replacements for functions in the C library which are known to be unsafe for usage with Cilk. The user can also include `cilk-compat.h` to make a Cilk program automatically use the Cilk replacements instead of the original C functions. The effect of `cilk-compat.h` is described in Section 8.5.

8.1 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.

8.2 Library functions

Currently, the Cilk library only consists of support for shared memory allocation and a random number generator. This section describes the available functionalities.

8.2.1 Shared memory support

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

meaning: Allocates `len` bytes on the shared-memory stack, and returns a pointer to the first byte.

[**Implementation note:** The current implementation of `Cilk_alloca` does not work properly from within C functions. See Section 6.14.1 for details.]

```
void *Cilk_malloc(size_t len);
void *Cilk_memalign(size_t len, size_t alignment);
void *Cilk_calloc(size_t nelem, size_t elsize);
void *Cilk_valloc(size_t size);
void *Cilk_realloc(void *s, size_t size);
void Cilk_free(void *ptr);
```

meaning: `Cilk_malloc` allocates `len` bytes on the shared-memory heap, and returns a pointer to the first byte. `Cilk_memalign` allocates `len` bytes on the shared-memory heap, aligned to at least an `alignment`-byte boundary. `Cilk_calloc` allocates space for `nelem` items of size `elsize` and initializes the memory to zero. `Cilk_valloc` allocates `size` bytes on the shared-memory heap, aligned to an operating system page boundary. `Cilk_realloc` takes a pointer originally obtained from one of these five allocator calls and changes its size. A pointer to the new storage is returned. Finally, `Cilk_free` frees a block of memory allocated by one of these five allocator calls.

8.2.2 Random number generator

```
extern int Cilk_rand(void);
extern void Cilk_srand(unsigned int seed);
```

Cilk provides its own random number generator, since we anticipate that the C random number generator will not work correctly. The usage is analogous to `rand()` and `srand()`.

8.2.3 Default libc routines

```
void *C_alloca(size_t len);
void *C_malloc(size_t len);
void *C_memalign(size_t len, size_t alignment);
void *C_calloc(size_t nelem, size_t elsize);
void *C_valloc(size_t size);
void *C_realloc(void *s, size_t size);
```

```
void C_free(void *ptr);
int  C_rand(void);
void C_srand(unsigned int seed);
void C_abort(void);
```

meaning: Aliases for standard (probably broken) libc routines.

8.2.4 Library version

```
int Cilk_parse_command_line(Cilk_options *options, int *argc,
                            char *argv[]);
void Cilk_init(Cilk_options *options);
int Cilk_run(cilk int (*func)(void*), void *arg);
void Cilk_finish(void);
```

meaning: `Cilk_parse_command_line` initializes `options` to the default Cilk values and then parses and removes standard Cilk arguments from `argc` (a pointer to the standard `argc` value) and `argv`. If the only desired behavior is to initialize `options`, pass 0 into `argc`. `Cilk_parse_command_line` returns 1 to indicate that the user requested help, 0 otherwise. `Cilk_init` starts idle worker threads according to the `options`. `Cilk_run` schedules Cilk function `func` according to the options specified on initialization. `func` is passed the argument `arg`. When `func` returns, `Cilk_run` returns the result of `func`. Note that `Cilk_run` is non-reentrant: It must never be called while another `Cilk_run` is in progress. `Cilk_finish` terminates the Cilk worker threads.

8.3 Macros

```
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.

8.4 Locks

```
void Cilk_lock_init(Cilk_lockvar lock);
void Cilk_lock(Cilk_lockvar lock);
void Cilk_unlock(Cilk_lockvar lock);
```

meaning: `Cilk_lock_init()` initializes a Cilk lock, which is of type `Cilk_lockvar`. `Cilk_lock()` and `Cilk_unlock()` acquire and release the lock pointed to by their argument. `Cilk_lock()` will not return until the lock is acquired.

8.5 Compatibility mode

Inclusion of the file `cilk-compat.h` causes the standard C library function names to be aliased to their Cilk equivalents. For example, with `cilk-compat.h` included, `malloc` is an alias for `Cilk_malloc`. See Section 6.14.1 for a complete list of aliased functions.

Chapter 9

The Nondeterminator

The Nondeterminator is a debugging tool to help users detect and locate data-race bugs in their Cilk programs [14, 10]. A ***data race*** occurs when two logically parallel threads access the same memory location while holding no locks in common, and at least one of the threads writes the location. The Nondeterminator runs a Cilk program with a given input serially and then examines the generated computation dag for data races. It is guaranteed to find a data race in the dag if and only if one exists. Such data races may not always correspond exactly to race conditions in the program, however, as the computation dag may change with scheduling. The Nondeterminator, therefore, is best thought of as a debugging tool, and not a program verifier. In this chapter we describe the functions and limitations of the Nondeterminator, as well as suggested ways to handle the limitations. This description includes a summary of available command-line options as well as linguistic extensions to the Cilk language.

For a quick overview of how to use the Nondeterminator, see Section 2.8.

9.1 The Nondeterminator's race reports

When the Nondeterminator finds a data race, it reports the memory address involved in the race, the variable names of the two competing accesses, and the line numbers and file names of those accesses. A typical report appears similar to the following:

```
---------------------------------------
Data race @ 0x2daa8
   'x' (data-race.cilk:17) with
   'x' (data-race.cilk:23)
---------------------------------------
```

To reduce the number of reports, the Nondeterminator will not report a race between two pieces of text in the code more than once, even if multiple executions of the same code result in multiple race conditions.

9.2 Memory allocation in the Nondeterminator

The Nondeterminator also correctly handles data races involving calls to `Cilk_malloc` and `Cilk_free`. That is, if a program accesses a memory location l in thread T_1 and frees that location in a parallel thread T_2, the Nondeterminator will report this as a race. (In some schedulings, location l could be freed, reallocated, and written to before T_1 is executed.) Such a race will appear similar to the following (taken from the example program `pllfree.cilk` in the `examples/nd` directory):

```
---------------------------------------
Free:  Data race @ 0xea828228
   `q' (pllfree.cilk:14) with
   `q' (pllfree.cilk:34)
---------------------------------------
```

This message will also be printed in the case where a program frees a memory location and then later accesses that memory location in serial. Although it may not be immediately apparent, this situation does represent a data race, since another thread (or the runtime system itself) could reallocate and write to that space in parallel.

As a side benefit, the Nondeterminator will print a message if the user's program accesses memory which has never been allocated, as in the following:

```
---------------------------------------
Access to non-allocated memory @ 0xea828304: `(t+1)' (pllfree.cilk:47)
---------------------------------------
```

In order to produce easy-to-understand race messages, the Nondeterminator implements memory allocation functions (see Section 6.14.1) without recycling memory. That is, once a portion of memory has been given out by a call to `Cilk_malloc`, it will never be returned by another call to `Cilk_malloc`, even if it has been freed in the meantime. Therefore, you may need to run the Nondeterminator version of the program with a larger heap size than the ordinary version (or use the `-recycle-mem` option).

9.3 The Brelly algorithm

While the Nondeterminator is generally an efficient debugging tool, it may run slowly for programs that make heavy use of locking. In that case, users may wish to run the Brelly algorithm (via the `-brelly` option), which detects violations of the "umbrella locking discipline" rather than data races. The ***umbrella locking discipline*** requires all accesses to any particular location within a given parallel subcomputation to be protected by a single lock. Subcomputations in series may each use a different lock, or even none, if no parallel accesses to the location occur within the subcomputation.

Every data race is a violation of the umbrella discipline, so the BRELLY algorithm still detects all data races in a Cilk computation, but the algorithm may produce additional reports for discipline violations that do not constitute data races.

The program `umbrella-violation.cilk` in the `examples/nd` directory of the Cilk distribution is an example of a program that contains an umbrella discipline violation but no data race. When run on that program with the `-brelly` option, the Nondeterminator produces the following report:

```
---------------------------------------
Umbrella discipline violation @ 0x2d27c: `x' (umbrella-violation.cilk:25)
     Access without lock `A' (0x2d280): `x' (umbrella-violation.cilk:17)
     Access without lock `B' (0x2d284): `x' (umbrella-violation.cilk:14)
---------------------------------------
```

Note that umbrella violations may consist of any number of accesses. The Nondeterminator first reports one access involved in the violation. For each lock held at the time of that access, the Nondeterminator reports another access that did not hold that lock. In other words, the Nondeterminator gives enough information to demonstrate that a discipline violation occurred.

In some simple cases, the Nondeterminator can recognize that an umbrella violation is in fact a data race, and reports it as such. It is still the case, however, that some umbrella violation reports may contain actual data races. It is up to the user to examine each report and determine if it constitutes a data race.

9.4 Limitations of the Nondeterminator

The Nondeterminator is guaranteed to detect a data race in a Cilk computation if and only if one exists. Data races in the computation sometimes may not correspond to race conditions actually acheivable by the program, however. The following is a list of the possible reasons for any such discrepancies, as well as the technical limitations of the Nondeterminator:

- *When the Cilk program run serially (on one processor) is already nondeterministic.* This situation may occur, for example, if the program's behavior is dependent on a random-number generator. In that case, the serial nondeterminism may change the computation, and so there may be data races in computations other than the one examined by the Nondeterminator.

- *When the order of lock acquires affects the control flow of the program.* This typically occurs when there are noncommutative critical regions or when locks are used to syncrhonize the program, and once again may change the generated computation dag. The Nondeterminator assumes that locks are used only for atomicity and not for synchronization, so it may consider threads to be "in parallel" when in reality they are ordered by locking protocols. The same

scenario may occur when the program references the Cilk scheduler variable `SYNCHED`. Thus, the Nondeterminator may sometimes report "false" race conditions. These false reports can be removed via user annotation of the code, as described in Sections 9.11 and 9.5.

- *When shared-memory accesses occur in a Cilk procedure, a user's C function, or a C library function, where the procedure or function definition has not been compiled by the Nondeterminator's* `cilk2c`. The Nondeterminator cannot track these accesses, and so may miss data races involving them. For this reason, the Nondeterminator issues warnings when external function calls are made, as described in Section 9.6. Users can manually instrument their code to specify how shared memory is accessed using the functions `Cilk_read_check` and `Cilk_write_check`, as is described in Section 9.10.

- *When memory is shared via OS threads rather than the Cilk runtime system.* The Nondeterminator only tracks memory dynamically allocated by `Cilk_malloc` or statically allocated in a file compiled by `cilk2c`. Memory that is shared via OS threads (see Section 6.13.3) will not be tracked, and races involving that memory will not be reported.

- *Inlets.* The Nondeterminator may report false race conditions in some situations involving inlets. See Section 9.7 for more details.

Note that the absence of data races does not guarantee that a program is deterministic. A program with no data races may still be nondeterministic if critical sections do not commute. Also, a data race itself may change the rest of the computation dag, so that once the Nondeterminator finds one data race, it may not find all other possible data races. The user is advised to run the Nondeterminator again after fixing any data race bugs found.

9.5 Fake locks and memory publishing

As detailed in this chapter, the Nondeterminator may sometimes produce "false" race messages. Users may wish to avoid these messages to prevent them from obscuring true race reports. One way to do this is to turn the Nondeterminator's checking off, using the pragmas described in Section 9.11. Turning checking off is dangerous, however, because true races might be missed. The Nondeterminator provides a few "safer" ways to remove false reports.

One such way is the use of ***fake locks***. Fake lock instructions are nops during ordinary execution, but are treated as regular locks by the Nondeterminator. `Cilk_fake_lock()` and `Cilk_fake_unlock()` both take a single variable of type `Cilk_lockvar`, and behave exactly like `Cilk_lock()` and `Cilk_unlock()`, respectively, during Nondeterminator execution. It is permissible, therefore, to use both fake lock and "real" lock instructions on the same lock.

When a false data race report is generated by the Nondeterminator, it can be eliminated by surrounding the offending accesses with a fake lock of the same lock; to the Nondeterminator, the accesses will no longer constitute a data race. Of course, the user must manually verify that there is no race, because fake locks will have no effect during ordinary Cilk execution.

If using the Brelly version of the Nondeterminator (see Section 9.3), you may need to surround several accesses with a single fake lock in order to comply with the umbrella locking discipline.

Unfortunately, fake locks may not always be practical, because there may be too many accesses to a memory location to surround them all with fake locks. We have found that there is one common scenario in which this occurs, which we call ***memory publishing***. Memory publishing occurs when a thread (atomically) adds an object to a global data structure, and then parallel threads access that object through the structure. To the Nondeterminator, this appears to be a data race, because it does not recognize that the read of the object is implicitly synchronized to come after earlier writes. The Nondeterminator provides the following macros to declare that this situation occurs:

```
PUBLISH(p);
PUBLISH_ARRAY(p, size);
```

`p` must be a *typed* pointer that points to the object being published. In the case of an array, the number of elements of the array is passed as the `size` parameter to `PUBLISH_ARRAY`. The meaning of these statements is that all prior accesses to the structure pointed to by `p` should be considered private accesses, and no more data races should be reported involving those accesses. There may still be data races reported on that memory, but only involving accesses that all occur after the `PUBLISH` statement is executed (in the Nondeterminator's serial order).

9.6 The Nondeterminator's compile-time warnings

Unless source code is run through the `cilk2c` compiler for the Nondeterminator, the Nondeterminator is unable to instrument the code to detect data races at runtime. Consequently, if a Cilk program spawns a Cilk procedure or calls a C function that is not defined in the same program, and the procedure or function contains a pointer argument, the Nondeterminator's `cilk2c` will issue a warning.

There are three cases in which warnings may be issued:

- an incomplete Cilk program in which a Cilk procedure defined elsewhere is spawned,
- an incomplete Cilk program in which a C function defined elsewhere is called,
- a complete Cilk program in which a C library function is called.

In all three cases, a warning will be generated only if the procedure or function is passed a pointer, since it is only by accessing shared memory that a data race can occur.

An example of this warning can be seen by compiling `queens-race.cilk`, which is an n-queens puzzle program that can be found in the `examples/nd` directory. The program uses the C library function `memcpy` for copying data from one location in memory to another, and `cilk2c` generates the following messages:

```
queens-race.cilk:42: Warning: (for the Nondeterminator) 'memcpy'
may access shared memory through its pointer arguments
queens-race.cilk:48: Warning: (for the Nondeterminator) 'memcpy'
may access shared memory through its pointer arguments
...
```

The `queens-race.cilk` code does in fact contain a data race involving calls to `memcpy`, but the Nondeterminator will not detect it, because the source code for `memcpy` was not compiled by `cilk2c`. In order to remedy this situation, the user can add calls to the Nondeterminator's instrumentation routines, as described in Section 9.10, to tell the Nondeterminator exactly what memory is accessed by `memcpy`.

9.7 Inlets

In addition to locks, Cilk provides inlets as another mechanism to provide atomicity. The Nondeterminator treats explicit user inlets as if they were regions of code surrounded by locks, and so does not report data races on shared memory accesses.

Implicit inlets, on the other hand, are unique because the Nondeterminator can determine whether the atomic accesses are in fact commutative. Thus the Nondeterminator will report a race bug between two implicit inlets that are not commutative, as in the following code:

```
x += spawn foo();
x *= spawn foo();
```

Since the Nondeterminator cannot determine whether an implicit inlet commutes with an explicit one, it will report the pair as a data race if they access the same memory location. This report can easily be avoided by the use of fake locks, as described in Section 9.5.

Cilk guarantees that updates done by inlets are atomic with updates executed by the parent procedure itself. The Nondeterminator, however, does not recognize this, and so reports a race condition. The following is an example:

```
x += spawn foo();
x += 1;
```

The Nondeterminator reports a race even though the updates to `x` are atomic and commutative. This once again can be fixed with the fake lock technique described in Section 9.5.

By default, the Nondeterminator does not consider any updates to floating-point variables to be commutative, and so will report races between implicit inlets that update floating point variables. To suppress these messages, use the `-float-comm` runtime option, which tells the Nondeterminator to assume that floating-point updates are commutative.

9.8 The Nondeterminator's `cilk2c` options

Besides the standard `cilk2c` options, the Nondeterminator's `cilk2c` provides the following additional option:

`-nd` Instrument reads and writes on shared memory.

9.9 The Nondeterminator's runtime options

Besides the standard Cilk runtime options (Section 2.2), a Cilk program linked with the Nondeterminator's runtime library accepts the following additional options:

`-brelly` Run the BRELLY algorithm for detecting umbrella discipline violations (see Section 9.3).

`-first-stop` Terminate the program and dump core when the first data race is found.

`-float-comm` Assume that floating-point updates are commutative when the Nondeterminator checks locations accessed by implicit inlets.

`-recycle-mem` Allow the Nondeterminator to reuse freed memory. This may cause some races to be missed, but saves on memory usage.

9.10 The Nondeterminator's library functions / macros

The following two library functions can be used in a Cilk program to inform the Nondeterminator of accesses to shared memory in C functions (or Cilk procedures) that have not been compiled with `cilk2c -nd`.

```
void Cilk_read_check(void *p, size_t size);
void Cilk_write_check(void *p, size_t size);
```

meaning: `Cilk_read_check` informs the Nondeterminator that the shared-memory region of `size`-bytes starting from `p` is read. `Cilk_write_check` informs the Nondeterminator that the same shared-memory region starting from `p` is written. These functions have no effect during ordinary Cilk execution (without the Nondeterminator).

The following two library functions implement fake locks for users to eliminate false reports from the Nondeterminator.

```
void Cilk_fake_lock(Cilk_lockvar lock);
void Cilk_fake_unlock(Cilk_lockvar lock);
```

meaning: `Cilk_fake_lock` and `Cilk_fake_unlock` behave exactly like `Cilk_lock()` and `Cilk_unlock()` respectively. These functions have no effect during ordinary Cilk execution (without the Nondeterminator).

The following two macros implement memory publishing for users to eliminate false reports from the Nondeterminator.

```
PUBLISH(p);
PUBLISH_ARRAY(p, s);
```

meaning: `PUBLISH` informs the Nondeterminator that all prior accesses to the structure pointed to by `p` should be considered private accesses, and these accesses won't consitute data races with future accesses. `PUBLISH_ARRAY` informs the Nondeterminator that all prior accesses to the array of size `s`, pointed to by `p`, should be considered private accesses, and these accesses won't consitute data races with future accesses. These macros have no effect during ordinary Cilk execution (without the Nondeterminator).

9.11 Pragmas and attributes

The following pragmas can be used in a Cilk program to inform the Nondeterminator's `cilk2c` that some code segments need not be instrumented, or warning on spawning procedures or calling functions with pointer arguments should be turned off. These pragmas are recognized by `cilk2c -nd`.

```
#pragma nd -
```

meaning: No instrumentation of reads and writes of shared memory is done onwards until the end of the program or until `#pragma nd +`.

```
#pragma nd +
```

meaning: Instrumentation of reads and writes of shared memory is resumed onwards until the end of the program or until `#pragma nd -`.

Wrapping `#pragma nd -` and `#pragma nd +` around a code segment tells `cilk2c` not to instrument shared-memory references in the code segment, so that the Nondeterminator will not check any shared-memory references in the code segment. For example,

```
#pragma nd -
cilk int foo()
{
    ...
}
#pragma nd +
```

the above Cilk code makes the Nondeterminator not check the procedure `foo()`. Note that calls to `Cilk_malloc` and `Cilk_free` are still instrumented within that region.

`#pragma ndwarn -`

meaning: No warning is issued on spawning procedures or calling function with pointer arguments onwards until the end of the program or until `#pragma ndwarn +`.

`#pragma ndwarn +`

meaning: Warning is resumed on spawning procedures or calling function with pointer arguments onwards until the end of the program or until `#pragma ndwarn -`.

Wrapping `#pragma ndwarn -` and `#pragma ndwarn +` around a code segment tells `cilk2c` not to issue warnings from the Nondeterminator.

The following attribute can be used in a Cilk program to inform the Nondeterminator that some global variables need not be checked. This attribute is recognized by `cilk2c -nd`.

`__attribute__((nocheck))`

meaning: This attribute can be attached to a global variable definition. No instrumentation is done on all accesses of this variable in the program.

The above attribute may be useful for running the Nondeterminator without monitoring some known race variables (so that the race warning on these variables will not be flagged). For example, `knapsack.cilk` in `examples/nd` has an intentional data race when independent threads atomically update the variable `best_so_far` containing the bound in its branch-and-bound search. To avoid checking on those variables known to have a race, you can attach `__attribute__((nocheck))` in the definition of those variables. For example,

```
int bar __attribute__((nocheck));
```

informs `cilk2c` not to instrument all accesses of the variable `bar` in the Cilk program.

Appendix A

Installation Instructions

Prerequisites

In order to install Cilk, download the latest cilk version from our web page at

`http://theory.lcs.mit.edu/~cilk`.

You need a recent version of `gcc` (we used `gcc-2.7.x`), the GNU linker[1] (part of the binutils package, we used version 2.9.1), and gcc must be configured so that it uses the GNU linker by default. Furthermore, GNU's `make` utility may be required to accept our Makefiles.

Note that there is a bug in the new release of gcc-2.8.1, which is currently being fixed by the GNU folks. Cilk does not work correctly with this version of gcc. We will announce on our web page when it is safe to use gcc-2.8.x.

How to install Cilk

You should get a working cilk installation by performing the following steps assuming the Cilk archive file is `Cilk5.2.tar.gz`.

- Copy the `Cilk5.2.tar.gz` into a directory on your system.
- Type `gunzip Cilk5.2.tar.gz` and `tar xvf Cilk5.2.tar`. This will create a directory named `cilk` and moves all of the Cilk software into this directory.
- To compile the Cilk system `cd cilk`, and run `configure` and then `make`. The default configuration will use one Unix process per process(or) requested with the `-nproc` option, when running a Cilk program. This version supports shared and private variables as well; cf. Section 6.13.1.

  Currently, Cilk runs on the following systems; use the given commands for installation.

[1] Actually, the GNU linker is only required for the process version of Cilk, not for the threaded version.

SparcV8/Solaris/Processes Cilk is installed appropriately on SparcV8 architectures using the default commands:

```
./configure ; make
```

SparcV9/Solaris/Processes An explicit option is required if optimized code for the SparcV9 architecture is desired:

```
./configure --with-sparc-V9 ; make
```

SparcV8/Solaris/Threads Uses one OS-thread per process(or) requested with the `-nproc` option when running a Cilk program.

```
./configure --with-os-threads ; make
```

SparcV9/Solaris/Threads Uses one OS-thread per process(or) requested with the `-nproc` option when running a Cilk program.

```
./configure --with-sparc-V9 --with-os-threads ; make
```

Pentium/Linux Cilk works with one Linux process per process(or) requested with the `-nproc` option when running a Cilk program. No special options are required.

```
./configure ; make
```

SGI/IRIX Cilk works with one IRIX-thread per process(or) requested with the `-nproc` option when running a Cilk program. No special options are required.

```
./configure ; make
```

Alpha/OSF Unix/Processes Cilk works with one OS-process per process(or) requested with the `-nproc` option when running a Cilk program. No special options are required.

```
./configure ; make
```

Alpha/OSF Unix/Threads Cilk works with one OS-thread per process(or) requested with the `-nproc` option when running a Cilk program.

```
./configure --with-os-threads ; make
```

- You can find cilk example programs in the example directory. It is advisable to run `testall` to check your installation.

Available `configure` options

You can control the installation using the following `configure` options:

- `--with-os-threads / --without-os-treads` use operating system threads instead of processes if available. Instead of using "heavy" processes as workers use operating system threads. See Chapter 4 for details.

- `--enable-stats / --disable-stats` enables collection of statistics. A runtime system compiled with –disable-stats is not capable of gathering any statistics. Therefore the runtime argument `-stats` is not available.
- `--enable-timing / --disable timing` enables collection of timing information, such as wall clock runtime, total work, etc. Timing must be enabled for critical path measurements.
- `--enable-debug / --disable-debug` enables Cilk consistency checks. We do not recommend to disable this feature.
- `--with-sparc-V9 / --without-sparc-V9` use compiler options for Sparc V9 processors.
- `--enable-library / --disable-library` enables the Cilk library version. The normal version of Cilk will be built as well. This option is only supported on the operating system thread versions of Cilk on some architectures.

You can change your installation by calling `configure` with the desired options and then `make` to recompile the whole system.

Appendix B

FAQ: Frequently Asked Questions

This appendix contains a list of frequently asked questions (and their answers) about Cilk.

Q: How do I get a copy of Cilk?

A: There is a web page for Cilk at `http://theory.lcs.mit.edu/~cilk`. The Cilk distribution and the manual (the document you are reading) are available for downloading from this web page.

Q: How do I complain about Cilk?

A: Send email to `cilk-support@supertech.lcs.mit.edu`.

Q: When I try to make the distribution, I get lots of errors from "make" about not being able to build certain targets.

A: Our `Makefile`s may require the use of GNU's `make` utility to work correctly, because they use GNU specific makefile syntax (automatic variables). Make sure that GNU's `make` is used instead of the default `make` on your system. Use the command `which make` to find out which `make` is used by default. To use GNU's `make` by default, add the path to the GNU binaries to the beginning of your path (the path to the GNU binaries is often `/usr/local/gnu/bin`), or just execute GNU's `make` directly (as in `/usr/local/gnu/bin/make fib`).

Q: `cilk2c` complains with the error message "identifier 'cilk_main' redeclared".

A: You forgot to declare your `main` routine to be a Cilk procedure: `cilk int main()`.

Q: The output from `cilk2c` will not compile. I get a slew of errors starting with "parse error before 'StackFrame'."

A: This is what happens when you do not include `cilk.h` at the top of your `.cilk` files. Add `#include <cilk.h>` to the top of the offending file and try again.

Q: On Linux, `cilk2c` complains about `__QI__` being not defined in `_G_config.h`.

A: Linux uses a `gcc` extension that `cilk2c` does not understand. `cilk.h` contains a workaround, but you must include `cilk.h` *before* any other include file.

Q: Why is `fib` so slow?

A: Normally, a `spawn` is about 4 times slower than a function call. Cilk becomes much slower, however, when the critical-path measurements are enabled. This is because Cilk reads the system clock at every thread boundary (i.e. at every `spawn`, `sync`, `return`). Some systems (Solaris) have fast user-level clock routines, but other systems (Linux) require a system call. Beware that the critical-path counter is enabled by default.

Q: `gcc` freezes during compilation. Why?

Q: My object files are very large. Why?

A: There is a side effect of our linking strategy which sometimes generates very large object files. Data which would normally go into the BSS segment (the uninitialized variable segment) is instead put into the DATA segment (the initialized variable segment). If you have a large uninitialized global array, our linking strategy will instead put it into the initialized data segment, and write out the initialization (all zeros) into the object file. To prevent this anomaly, you must allocate large arrays from within your code using `malloc()`.

Q: The linker fails on Solaris, complaining about an unknown `-T` option.

A: In order to operate properly, Cilk needs certain regions of memory that are shared among processors, and certain that are private. Although you might expect this to be a common problem, there is no standard way to do it.

The solution to adopt was to use the GNU linker. The GNU linker allows you to define your own memory map, including new segments of memory that can be mapped shared or privately. This is the meaning of the `-T` option that caused the error. Solaris's linker, surprisingly, does not have any comparable feature.

You have two options in order to solve your problem. The first solution is to edit `Makefile.common` and select the solaris-threads architecture instead of solaris. In this way, Cilk runs in an all-shared address space. You cannot have private data, and in particular the `testall` program will not work as is.

The second solution is to install the GNU linker. Notice that, by default, and for reasons that escape me, `gcc` *does not* use the GNU linker by default, even if

it is present. You need to explicitly compile `gcc` with this option, as explained in the gcc installation notes.

Q: My program halts with the error "fatal error: Cilk malloc() memory exhausted, can't alloc 134217792 bytes".

A: You are trying to `malloc()` more memory than the system will currently let you allocate. Use the `-heap` Cilk option to control how much heap is originally allocated for your use. For instance, putting `-heap 140` on the command line would reserve 140 megabytes of memory for the user heap, which should be enough for the example above if it is the only large `malloc()` in the program. The default amount of heap is currently set at 64 megabytes.

Cilk's `malloc()` is most space-efficient when you ask for an amount of memory that is a power of two. Allocating other sizes may cause Cilk to waste up to half of the heap because of the padding strategy `malloc` uses.

Q: `myprog -myopt -nproc 10` does not work. Why?

A: Cilk options must come before program-specific options in the command line. Use `myprog -nproc 10 -myopt`. Rationale: in this way an existing C program does not need to be Cilk-aware.

Q: My program works fine on one process under Cilk, but causes a segmentation fault on more than one process.

A: Although there are many bugs that can cause this problem, there are a few specific causes to be aware of. All variables in `.c` files are `private`, which means that they are not universally accessible when you run with more than one process. If you store a pointer in such a variable, and then try to read and dereference it on another processor, you will cause a segmentation fault. The fix is to make sure all files you are compiling that have global variables in them are `.cilk` files. It may be difficult to do this, however, because some of the code the user is linking may not be under his control. For example, it may not be practical to recompile an `X` window toolkit under Cilk. In these cases, the only solution we have for now is to make sure all calls to such a toolkit are contained within one thread so that pointers to private memory will not be migrated. Running Cilk with OS threads instead of processes may also solve this problem.

Q: The critical path measurement yields an unexpected value of zero.

A: Critical path measurements require that the `CILK_CRITICAL_PATH` flag is set in the application's `Makefile` when compiling the application. Also, some platforms do not support critical path timing at all because there is inadequate

support for high-resolution timing. In our current release, the IRIX port cannot measure critical paths because of this limitation.

Q: (Threaded versions) I can't work with an `-nproc` of more than X - Cilk complains and then exits.

A: Try lowering the default stack size with a command like "`limit stacksize 1M`". The system needs to allocate this much virtual address for each worker thread, and all the allocations must fit within the process' virtual address space (a serious problem for 32-bit systems). You can see what the current stack limit is set at with the command "`limit stacksize`".

Appendix C

People

The following people contributed to the development of Cilk 5.2:

Guang-Ien Cheng
Don Dailey
Mingdong Feng
Matteo Frigo
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 spend 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 D

Copyright and Disclaimers

Copyright © 1998 Massachusetts Institute of Technology

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to use, copy, modify, and distribute the Software without restriction, provided the Software, including any modified copies made under this license, is not distributed for a fee, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of the Massachusetts Institute of Technology shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from the Massachusetts Institute of Technology.

Inquiries about obtaining commercial licenses for Cilk technology should be sent to Charles E. Leiserson, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139; email: `cel@mit.edu`, phone: 617-253-5833, fax: 617-253-0415.

Bibliography

[1] Selim G. Akl and Nicola Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11), November 1987.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *In proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 28–July 2 1998.

[11] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, April 1965.

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

[13] 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.

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

[15] Leonardo Pisano (Fibonacci). Liber abbaci, 1202.

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

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

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

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

[20] 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.

[21] 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`.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] Robert Sedgewick. Implementing quicksort programs. *Comm. ACM*, 21(10), October 1978.

[29] Sivan Toledo. Locality of reference in lu decomposition using partial pivoting. Available at `http://theory.lcs.mit.edu/~sivan/029774.ps.gz`.

[30] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

Index

aborting work, 19–21, 63
`abort` keyword, 19–21, 60, **63**
algorithmic multithreaded language, 27
`alloca` function, 13, **69**
architecture-specific issues, 48–49
atomicity, 18, 66
    locks, *see* locks
    race, *see* data race
attributes
    `((nocheck))`, **86**
    porting Cilk, 46

Barnes-Hut n-body code example, 32
barrier, 8
`_BEFORE_RETURN_FAST` macro, 53
`_begin_Cilk_shared` macro, 49
Brelly, 25, 79
Bucket Sort example, 32–33
bugs, 90–93
    `cilk2c`, 73
    reporting, i

cactus stack, **12–13**
`calloc` function, **69**
`C_alloca` function, **69**
`C_calloc` function, **69**
`C_free` function, **69**
Checkers example, 35
Cilk
    Support, i
`Cilk_active_size` variable, **74**
`Cilk_alloca` function, 13, 64, 69, **75**
`Cilk_assert` macro, **76**
`Cilk_calloc` function, 64–65, 69, **75**
`cilk-compat.h` header file, 11, 12, 69, 74, **77**
`CILK_CRITICAL_PATH` option, **67–68**, 92–93
`CILK_DIR` option, 9
`Cilk_fake_lock` function, **85**
`Cilk_fake_unlock` function, **85**
`Cilk_fence` function, **48**
    shared memory, 15–16
`.cilk` file extension, **9**
`Cilk_finish` function, 23, 76
`Cilk_free` function, 13, 64–65, 69, **75**
`Cilk_get_wall_time` function, **67–68**
`cilk.h` header file, 7, 74
`Cilk_init` function, 22, 76
`cilk` keyword, 7, 60, **60**
`cilk-lib.h` header file, 11, 12, 74
Cilk libraries, 74–77
    functions, **74–76**
    global variables, **74**
    libc routines, 75–76
    locks, 77
    macros, 76
    Nondeterminator, 84–85
    random number generator, 69, **75**
    shared memory, 75
`Cilk_lock` function, 16, 65, **77**
`Cilk_lock_init` function, 16, 65, **77**
`Cilk_lockvar` type, 16, 65, **77**
`cilk_main` redeclared, 90
`Cilk_malloc` function, 11, 13, 64–65, 69, **75**, 77
`Cilk_memalign` function, 69, **75**
`Cilk_membar_StoreLoad` function, **48**
`Cilk_membar_StoreStore` function, **48**
`cilk-nd.h` header file, 49
`Cilk_options` type, 22, 76
`Cilk_parse_command_line` function, 22,

76
Cilk procedure, **7**–**8**
`Cilk_rand` function, 69, **75**
`Cilk_read_check` function, **84**–**85**
`Cilk_realloc` function, 64–65, 69, **75**
`Cilk_run` function, 22–23, 76
Cilk-Safe, **11**, 69
`Cilk_shared` segment, 46
Cilksort example, 34–35
`Cilk_srand` function, 69, **75**
`Cilk_time` type, **67**–**68**
`Cilk_time_to_sec` function, **67**–**68**
`cilk2c`, 9, **71**–**73**
  bugs, 73
  command line, 71–73, 84
  internals, 73
`Cilk_unlock` function, 16, 65, **77**
Cilk-Unsafe, **11**
`Cilk_user_critical_path` variable, **67**–**68**
`Cilk_user_work` variable, **67**–**68**
`Cilk_valloc` function, 64–65, 69, **75**
`Cilk_wall_time_to_sec` function, **67**–**68**
`Cilk_write_check` function, **84**–**85**
C libraries
  Cilk-Safe functions, 11
  compatibility, 69
  libc routines, 75–76
`clone`, 46
`C_malloc` function, **69**
`C_memalign` function, **69**
command line
  `cilk2c`, 71–73, 84
  runtime, 10–11, 84
commercial licensing, 96
compiling Cilk, 51–55
  `cilk2c`, **71**–**73**
computation, model of, 25–27
consistency models, 16
`C_rand` function, **69**
`C_realloc` function, **69**
critical path, 4, **26**, 67–68, 92–93
critical section, **16**
`C_srand` function, **69**
`C_valloc` function, **69**

dag consistency, 17–18
data-parallel, 3
data race, 4, **14**–**16**, **23**, 66, 78, 86, *see also* Nondeterminator
deadlock, 65–66
debugging, 23
  data race, 4, **14**–**16**, 66, 86, *see also* Nondeterminator
`DECLARE_PRIVATE_VARIABLE` macro, 45
`DECLARE_SHARED_VARIABLE` macro, 45
Dense LU Factorization example, 39
deque, **51**

`_end` macro, 49
`_end_Cilk_shared` macro, 49
`_etext` macro, 49
examples, 30–33
  Barnes-Hut n-body code, 32
  Bucket Sort, 32–33
  Checkers, 35
  Cilksort, 34–35
  Dense LU Factorization, 39
  Fast Fourier Transform, 35
  Fibonacci, 35–36
  Game, 36
  Heat Diffusion, 36
  Kalah, 37
  Knapsack, 37–38
  Knary, 38
  LU with partial pivoting, 39–40
  Magic Squares, 40
  Matrix Multiplication, 40–41
  N-Queens, 41
  Sparse Cholesky Factorization, 33–34

Fast Fourier Transform example, 35
`_fdata` macro, 49
Fibonacci example, 35–36
file extension
  `.cilk`, **9**
file I/O, 69–70

frame, **51**
_frame variable, 52
free function, 13, **69**
frequently asked questions
  bugs, 90–93
functions
  Cilk libraries, **74–76**

Game example, 36
_G_config.h header file, 91

header files
  cilk-compat.h, 11, 12, 69, 74, **77**
  cilk.h, 7, 74
  cilk-lib.h, 11, 12, 74
  cilk-nd.h, 49
  _G_config.h, 91
HEAP_ACCESSOR macro, 49
HEAP_LOCKLISTS macro, 49
HEAP_LOCKS macro, 49
heap region, **49**
Heat Diffusion example, 36

I/O, 69–70
implicit inlets, **18**
implicit sync, 8
_INIT_FRAME macro, 53
initial thread, **8**
inlet keyword, 60, **62–63**
inlets, 17–19
  atomicity, 18
  defining, 62–63
  implicit, **18**
  invoking, 63
installation, 87–89

Kalah example, 37
keywords, 59–60
  abort, 19–21, 60, **63**
  cilk, 7, 60, **60**
  inlet, 60, **62–63**
  private, 60, **66–67**
  return, **62**
  shared, 60, **66–67**
  spawn, 8, 60, **61**
  sync, 8, 60, **62**
  SYNCHED, 21–22, 60, **64**
Knapsack example, 37–38
Knary example, 38

language reference, 59–70
libc routines, 11, 69, 75–76
library version, 22–23, 76
  Cilk_finish, 23, 76
  Cilk_init, 22, 76
  Cilk_options, 22, 76
  Cilk_parse_command_line, 22, 76
  Cilk_run, 22–23, 76
  enabling, 89
licensing, 96
locks, 77
  porting Cilk, 47
  using, 16, 65–66
LU with partial pivoting example, 39–40

Magic Squares example, 40
main function, 59
Makefile, 9, 30
  options, 9, 67, 92–93
  problems, 90
malloc function, 13, **69**, 77
Matrix Multiplication example, 40–41
memalign function, **69**
memory allocation, 12–13, 64–65, 69, **75**
memory consistency, *see* shared memory
message-passing style, 3
microscheduling, 53–55
mmap function, 45–47
model of computation, 25–27

N-Queens example, 41
nanoscheduling, 51–53
Nondeterminator, 23–25, 78–86
  accessor shadow space, **49**
  attributes, 86
  Brelly, 25, 79
  command line, 84

compile-time warnings, 82
data race, 4, **14–16**, 66, 86
fake lock, 81–82
heap region, **49**
info shadow space, **49**
inlets, 83
library functions, 84–85
limitations, 80
lockers shadow space, **49**
locks shadow space, **49**
memory allocation, 79
memory publishing, 81–82
porting, 49–50
pragmas, 85–86
race reports, 78
shadow spaces, 49–50
static region, **49**
var shadow space, **49**

open issues, 69–70
OS-thread, 68
overhead, 55

porting Cilk, 45–50
architecture-specific issues, 48–49
attributes, 46
locks, 47
memory model, 45–47
Nondeterminator, 49–50
shared memory, 47–48
pragmas
`lang +/-C`, **60**
`nd +/-`, **85–86**
`ndwarn +/-`, **86**
`private` keyword, 60, **66–67**
procedure, **7–8**
`PUBLISH` function, **85**
`PUBLISH_ARRAY` function, **85**

`__QI__`, 91

race, 4, **14–16**, 66, 86, *see also* Nondeterminator
`rand` function, **69**
random number generator, 69, **75**
`realloc` function, **69**
release notes, i
`return` keyword, **62**

scheduling, 51–56
aborting work, 19–21, 63
atomicity, 18
critical path, 4, **26**, 67–68, 92–93
deque, **51**
frame, **51**
initial thread, **8**
inlets, *see* inlets
interaction with, 21–22, 64
microscheduling, 53–55
model of computation, 25–27
nanoscheduling, 51–53
overhead, 55
page faults, 4
slow version, **53–55**
spawn tree, **8**
statistics, 4, 10–11, 26–27, 67–68, 92–93
successor threads, **8**
thief, **53**
thread, **8**
timing, 4, 10–11, 26–27, 67–68, 92–93
victim, **53**
weak memory models, 56
work, 4, **26**, 67–68
work-stealing, **26**
segmentation fault, 92
`Self` variable, **74**
serial elision, 7
`_SET_RESULT` macro, 53, 55
`SHADOW_INFO` macro, 50
shadow spaces, 49–50
`SHADOW_SPACE_SHIFT` macro, 50
`SHADOW_VAR` macro, 50
`shared` keyword, 60, **66–67**
shared memory
allocation, 12–13, 64–65, 69, **75**
cactus stack, **12–13**
`Cilk_fence`, 15–16, **48**

consistency models, 16
dag consistency, 17–18
functions, 75
porting Cilk, 45–48
weak memory models, 56
`shmget` function, 47
slow version, **53**–**55**
socket I/O, 69–70
Sparse Cholesky Factorization example, 33–34
`spawn` keyword, 8, 60, **61**
spawn tree, **8**
`srand` function, **69**
`StackFrame` parse error, 90–91
`STATIC_ACCESSOR` macro, 49
`STATIC_ARENA_END` macro, 49
`STATIC_ARENA_START` macro, 49
`STATIC_LOCKLISTS` macro, 49
`STATIC_LOCKS` macro, 49
static region, **49**
statistics, 10–11, 26–27, 67–68
critical path, 4, **26**, 67–68, 92–93
work, 4, **26**, 67–68
storage allocation, 12–13, 64–65, 69, **75**
successor threads, **8**
super-linear speedup, 38
superlinear speedup, 41
Support, i
`SYNCHED` keyword, 21–22, 60, **64**
`sync` keyword, 8, 60, **62**
implicit, 8
`_SYNC` macro, 55

thief, **53**
thread, **8**
timing, 10–11, 26–27, 67–68
critical path, 4, **26**, 67–68, 92–93
work, 4, **26**, 67–68

umbrella locking discipline, **79**
`USE_PRIVATE` macro, 45
`USE_SHARED` macro, 45

`valloc` function, **69**
variables
global, **74**
shared vs. private, **66**–**67**
victim, **53**

weak memory models, 56
work, 4, **26**, 67–68
work-stealing, **26**

`_XPOP_FRAME_RESULT` macro, 53