Cilk-5.1 (Beta 1) Reference Manual[1]

Supercomputing Technologies Group
MIT Laboratory for Computer Science

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

September 10, 1997

[1]Cilk is a trademark of the Massachusetts Institute of Technology. The Cilk project is supported in part by DARPA Grant N00014-94-1-0985.

Release Notes

Cilk-5.1 (Beta 1)

- Many features were added to the Nondeterminator, including proper handling of `malloc` and `free`. The former "luxurious" version, recognized by the `.ndl` extension, is now the default version, using the `.nd` extension.
- User-level locks have been added.

Cilk-5.0 (Beta 1)

- New, faster, more portable run-time system.
- Besides the Cilk-5 base package, the full Cilk-5.0 package contains the Nondeterminator, a novel race detection tool for debugging. The Nondeterminator comes with a modified `cilk2c` and runtime system.
- Removed penetrable barriers, including the `resume` keyword. Removed `SPAWN_COUNTER` keyword. We do not expect to support these features in the future.

Cilk-4.1 (Beta 1)

- We are now using the GNU linker, which allows us to control the segments into which shared and private variables go. I/O now works correctly without crashing. Many `libc` functions, although probably not functionally correct, do not crash.

Cilk-4.0 (Beta 1)

- New lazy scheduler: it's fast!
- All of the Cilk-1 style code (explicit continuation passing) has been removed from the language. Also gone are `byvalue` arguments and a whole host of other hackery.
- Shared memory support is now transparent, and so the `shared` keyword, along with `register shared` and `page_aligned` keywords, have been removed. All user memory is now shared and dag consistent.

- Inlets, along with support constructs like `abort` and `resume` (penetrable barriers).
- An implementation on an SMP using shared memory hardware directly, instead of implementing a message-passing protocol on top of shared memory and using polling as in an MPP.

Contents

Part I — Tutorial

Chapter 1

Introduction

This document describes Cilk-5.1, a language for multithreaded parallel programming based on ANSI C. Cilk is designed for computations with dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style. Divide-and-conquer algorithms and tree search are examples of computations that are particularly well suited to Cilk. (For example, our group has developed three world-class chess programs, ⋆Tech [20], ⋆Socrates [18], and Cilkchess.) Besides search, we have experimented with 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-5.1?

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-developers@theory.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, 19].

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 [21] and which used work-stealing as a general scheduling policy to improve the load balance and locality of the computation [15]. 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, 19]. 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 as the scheduling abstraction. 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 [13], which helps Cilk programmers to localize determinacy-race bugs in their code.

To date, prototype applications developed in Cilk include graphics rendering, protein folding [25], backtracking search, N-body simulation, and dense and sparse matrix computations. Our largest application effort is a series of chess programs. The previous program, ⋆Socrates [18], finished second place (before Deep Thought) 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. The current 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 two MIT Ph.D. theses [3, 19] contain more detailed descriptions of the foundation and history of early Cilk versions.

1.3 About this manual

This manual is primarily intended for users who wish to write Cilk application programs. However, 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 and introduces various features of the Cilk language. Some of the example programs that are included in the Cilk distribution are discussed in Chapter 3.

Chapters 4 and 5 of Part II describe internals of the Cilk-5.1 runtime system and compiler. This part should be useful, if you want to know about the implementation of Cilk, or want to port Cilk.

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. Furthermore, 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), rather than distributed-memory machines. Section 2.1 illustrates the Cilk extensions to ANSI C through the example of computing Fibonacci numbers. Section 2.2 describes the computational model supported by Cilk. Section 2.3 then steps through the process of writing, compiling, and running simple Cilk programs. Section 2.4 describes rules for how Cilk and C interact. Section 2.5 introduces the issues in programming Cilk with shared memory, and Section 2.6 shows how memory can be allocated. Section 2.7 treats some of Cilk's more advanced language features, including constructs for aborting already-spawned work. Section 2.8 gives the recommended strategy for debugging Cilk code, along with an explanation on how to use the Nondeterminator debugging tool.

2.1 The Cilk extensions to ANSI C

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

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

One of the simplest examples of a Cilk program is a recursive program to compute the nth Fibonacci number. A C program to compute the nth Fibonacci number is shown in Figure 2.1(a), and Figure 2.1(b) shows a Cilk program that does the same computation in parallel. 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 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 completes before all of its children terminate.

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

2.2 The Cilk model of multithreaded computation

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

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

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

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

        return (x+y);
    }
}

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

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

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

(a)

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

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

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

        sync;

        return (x+y);
    }
}

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

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

    sync;

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

(b)

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

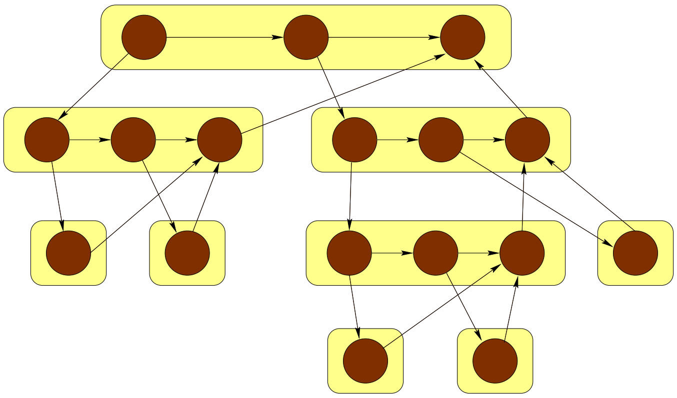

Figure 2.2: The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates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

A Cilk program execution consists of a collection of ***procedures***,[1] each of which is broken into a sequence of nonblocking ***threads***. In Cilk terminology, a ***thread*** is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (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***, as is illustrated in Figure 2.2.

A correct execution of a Cilk program must obey all the dependencies in the dag, since a thread cannot be executed until all the threads on which it depends have completed. These dependencies form a partial order, permitting many ways of scheduling the threads in the dag. The order in which the dag unfolds and the mapping of threads onto processors are crucial decisions made by Cilk's scheduler. Every active procedure has 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

[1] Technically, procedure *instances*.

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

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

Cilk's work-stealing scheduler executes a Cilk computation on P processors in time $T_P \leq T_1/P + O(T_\infty)$, which is asymptotically optimal. Empirically, the constant factor hidden by the big O is often close to 1 or 2 [6], and the formula

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

is a good approximation of runtime. (This model assumes that the parallel computer has adequate bandwidth in its communication network.) This performance model can be interpreted using the notion of ***average parallelism***, which is given by the formula $\overline{P} = T_1/T_\infty$. The average parallelism is the average amount of work for every step along the critical path. Whenever $P \ll \overline{P}$, meaning that the actual number of processors is much smaller than the average parallelism of the application, we have equivalently that $T_1/P \gg T_\infty$. Thus, the model predicts that $T_P \approx T_1/P$ and the Cilk program is guaranteed to run with almost perfect linear speedup. The measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs over the entire range of possible parallel machine sizes. Cilk provides automatic timing instrumentation (see Section 6.12.2) that 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

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

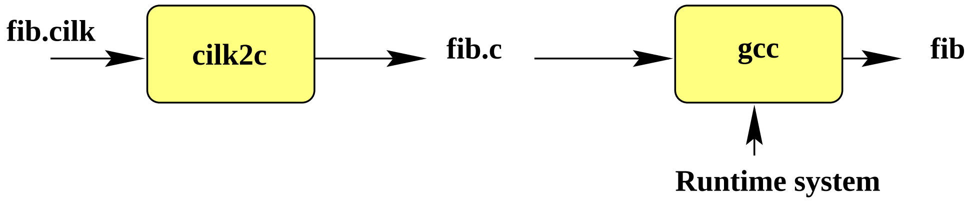

Figure 2.3: Compiling the Fibonacci program. Cilk code is passed through the `cilk2c` compiler, a type-checking preprocessor which generates C output. This output is then compiled with `gcc` and linked with the Cilk runtime library.

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, as we shall see in Section 2.6.1, Cilk also makes a guarantee about the amount of stack space used by a program.

2.3 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
```

[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.

to make the executable. To run the program, type

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

to execute `fib` on 4 processors with an argument of 30.

At the end of the execution, you should see a printout similar to the following:

```
fib(30) = 832040 on 4 processors
```

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:

```
fib(30) = 832040 on 4 processors
Wall-clock running time on 4 processors: 2.451818 s
Total work = 9.605744 s
Total work (accumulated) = 7.223940 s
Critical path = 877.492000 us
Average parallelism = 8232.485758
```

After the output of the program itself ("`fib(30) = 832040 on 4 processors`"), the runtime system reports several useful statistics collected during execution. The first line of statistics gives the number of active processors and the overall execution time in seconds. The second line reports the total number of seconds during which processors were executing Cilk code.[4] Note that the difference between the total work divided by the number of processors and the overall execution time is due to the overhead of Cilk's scheduler. The fourth line reports the critical-path length of the computation in seconds, but only if you have enabled this instrumentation during compilation by setting `CILK_CRITICAL_PATH = 1` in the `Makefile` of the example directory. The average 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.

Here is another example:

```
> fib -nproc 4 -stats 5 30
Wall-clock running time on 4 processors: 2.114724 s
Total work = 8.394466 s
Total work (accumulated) = 6.158160 s
Critical path = 530.855000 us
Average parallelism = 11600.456695

SUMMARY OF TIME SPENT BY EACH PROCESSOR:
```

[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.

```
 PN     Working        Total
---------------------------
  0    2.093347    2.096502
  1    2.092310    2.096052
  2    2.093485    2.094979
  3    2.091995    2.095461
---------------------------
TOT    8.371138    8.382994

SUMMARY OF TIME STATISTICS:

Total execution time: 8.382994s / wall 8.391091s (entered 4 times)
Working time: 8.371138s / wall 8.374485s (entered 109 times)
Waiting for locks: 0.004513s / wall 0.005767s (entered 1803 times)
Stealing: 0.007307s / wall 0.010912s (entered 876 times)
Returning: 0.000785s / wall 0.000922s (entered 63 times)
Aborting: 0.000002s / wall 0.000002s (entered 1 times)
```

Increasing the statistic level to 5 produces a table that reports on the total time used by each processor, and the portion of time spent executing the work of the program. The last line shows the sum of overall work time and total time. Additionally, at statistics level 5, a table showing the time spent on various runtime system operations is printed.

The critical path of the Fibonacci program is very short ($T_\infty \approx 0.0005$ seconds compared to $T_1/P \approx 8.4/4 \approx 2.1$ seconds). The Cilk performance model Equation (2.1) thus predicts near linear speedup over one-processor execution time. To verify that linear speedup really does occur, you can run `fib` on various numbers of processors (using the `-nproc` option described next) and compare their performances.

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

`-help` List available options.

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

`-stats` l Set statistic level to l. 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. 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`.

For example, one can invoke the `fib` program by

```
> fib -nproc 8 30
```

The runtime system interprets and strips away `-nproc 8`, so that the first and only option seen by `main` is `30`.

If you do not require any of the statistics that the runtime system computes, you can disable statistics gathering altogether by recompiling the runtime system with different compile-time flags. Without statistics gathering, the runtime system will be somewhat faster. See Section 2.7.4 for details.

2.4 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 C library was not designed for Cilk, and thus many library functions will 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 6.13.1.

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 6.13.1 for details.

2.5 Shared memory

Cilk supports shared memory. Sharing occurs when a global variable is accessed by procedures operating in parallel, but sharing can also arise indirectly from the passing of pointers to spawned procedures, allowing more than one procedure to access the object addressed by the pointer. (Cilk supports the same semantics for pointer passing as C. See Section 2.6.) 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 2.4 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 2.5 shows a slightly modified version of the Cilk procedures in Figure 2.4. 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 *determinacy 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 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.

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

```
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 2.4: 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 2.5: Nondeterministic behavior may result from shared access to the variable `x` when `x` is updated.

In some circumstances you may find you need to cope with the intricacies of sharing. At a minimum, any implementation of Cilk supports a relaxed consistency model called ***dag consistency*** [4, 5]. In addition, a Cilk implementation on a particular machine architecture generally allows you to assume (at your own risk) the stronger shared-memory semantics of that machine.

Cilk's dag-consistent shared memory provides you with a relaxed consistency model. The intuitive idea behind dag consistency is the following:[6]

> *A thread may see any value for a variable which is consistent with some serial execution order of the computation dag of threads.*

The serial orders seen by two different threads may differ, however, depending on the way the program is scheduled for execution. Moreover, the serial orders seen by a single thread for two different variables may differ as well.

The authors of Cilk anticipate that stronger consistency models will be supported in the future for specific circumstances, but that dag consistency will continue to be the default shared-memory semantics for Cilk. The advantage of supporting dag consistency is that operations on shared memory are fast, both empirically and theoretically, even on multiprocessors with distributed memory.

If the machine on which you run Cilk supports a stronger consistency model, you are free to use it. As an example, on an SMP, you can use the locking and unlocking

[6]This definition is for intuition only, since it is not well defined for certain nondeterministic programs. See [4] for a proper definition of dag consistency.

primitives described in Section 6.12.3 to update regions of memory atomically. Of course, you should be aware of the problems of deadlock that may occur, and your code will not be portable to machines that do not support the same model. But sometimes, the performance of a code on a particular machine outweighs the importance of portability and simplicity, so although we officially discourage the use of locking protocols, we begrudgingly support them. As an aid to portability, Cilk also 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. Judicious use of `Cilk_fence()` can often yield faster code than can explicit locking.

2.6 Storage allocation

Like C, Cilk has 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.

2.6.1 Stack memory

Cilk uses a ***cactus stack*** [23] for stack-allocated storage, such as is needed for procedure-local variables. As is shown shown in Figure 2.6, 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 2.6.

Cactus stacks in Cilk have essentially the same limitations as ordinary C stacks [23]. For instance, a child procedure cannot return to its parent a pointer to an object that it has allocated, since the object will be deallocated automatically when the child returns. Similarly, sibling procedures cannot reference each other's local variables. Just as with the C stack, pointers to objects allocated on the cactus stack can only be safely passed to procedures below the allocation point in the call tree.

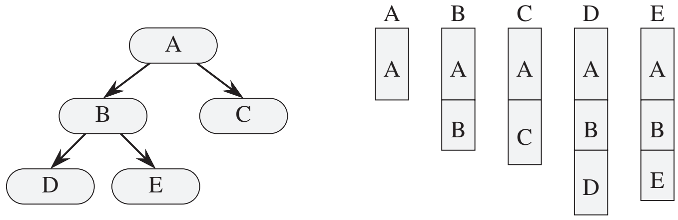

Figure 2.6: A cactus stack. Procedure A spawns B and C, and B spawns D and E. The left part of the figure shows the spawn tree, and the right part of the figure shows the view of the stack by the five procedures. (The stack grows downward.)

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 6.13.1.

2.6.2 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.

2.7 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. 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`. Section 2.7.4 describes some of

```
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 2.7: Using an inlet to compute the nth Fibonnaci number.

the compilation options of application programs and the Cilk runtime system. Finally, Section 2.7.5 briefly discusses nonstandard features for Cilk that are available in some, but not all, installations, as well as planned features that did not make it into the current release.

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 [11].

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

Figure 2.7 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()`.

```
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 2.8: Using implicit inlets to compute the nth Fibonnaci number.

Since an inlet is like a C function, it obeys the restrictions of C functions (see Section 2.4) 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 2.8 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 2.9 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 2.10.

The procedure `foo` spawns two children: `baz(17)` and `baz(28)`. The question is, when `baz(17)` returns, and the inlet executes the `abort` statement, is `baz(28)` aborted? The answer is, it depends. If `baz(28)` has already been spawned but has not yet completed, then it is aborted. If `baz(28)` has not been spawned, then it will be spawned and will not be aborted. In other words, `abort` *only aborts extant children, not future children.* If you want to prevent future children from being spawned, you should set a flag in the inlet indicating that an `abort` has taken place, and then test that flag in the procedure before spawning a child. Since threads that belong to the same procedure execute atomically with respect to each other (including the thread

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

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

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

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

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

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

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

    /* 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 2.9: 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 2.10: Illustration of an anomaly with the use of `abort`.

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

The variable `SYNCHED` is actually more like a macro than an honest-to-goodness variable. The Cilk type-checking preprocessor [22] actually produces two "clones" of each Cilk procedure: a "fast" clone that executes common-case serial code, and a "slow" clone that worries about parallel communication. In the slow clone, which is rarely executed, `SYNCHED` inspects Cilk's internal state to determine if any children are outstanding. In the fast clone, which is almost always executed, however, the value of

[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 2.11: An illustration of the use of `SYNCHED` to save storage and enhance locality.

`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 Makefile flags

Makefile options can be used to enable and disable features of Cilk. Makefile options come in two flavors, per-application options and global options. Per-application options are controlled from within the application's makefile, an example of which is the `Makefile` in the `examples` directory of the distribution. Global options are controlled from within the file `Makefile.common`, found in the top level directory of the distribution.

Each of these flags is described below. By default, all of these flags are on, meaning that the feature they control is enabled. By disabling some of the features that an application is not using, it is possible to speed the application up. We do not expect that most applications will see much speedup, but some (especially those that do not have much work per thread) may benefit significantly. If timer calls are expensive on your system, turning `CILK_CRITICAL_PATH` off might improve running times a lot. If you change any flags in your application's makefile, be sure to recompile the entire application. If you change any of the flags in `Makefile.common`, be sure to recompile *both* the Cilk runtime system and any applications you are using. The runtime system can be recompiled by executing `make` in the `rts` subdirectory of the distribution.

Per-application options

`CILK_CRITICAL_PATH` This flag enables different levels of critical path measurements. Value `0` disables critical path measurements, and should be used if the measurement overhead presents an obstacle to high performance. For program development, the value `1` is highly recommended, because the critical path as well as average parallelism measures are invaluable aids for performance tuning.

Global options

`CILK_DEBUG` If set to `1` this flag enables debugging within the Cilk runtime system.

`CILK_STATS` If this flag is set to `1`, statistical information will be gathered, such as the number of threads generated and the number of steals. The amount of information printed after a program run is determined by the runtime argument `-stats`.

`CILK_TIMING` This flag enables time measurements such as wall clock runtime, total work, etc. This flag must be enabled for critical path measurements to work.

2.7.5 Nonstandard and unimplemented features

Some releases of Cilk contain features that are not supported in all releases. These "nonstandard" features include private global variables, timers, and locks. See Section 6.12 for a description of these features.

We hope to put additional features into Cilk in the near future. Unimplemented features include the following:

- 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.
- A `cilk_asm` feature which allows C code to be passed directly through `cilk2c` to the C compiler.
- 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.

- A parallel file I/O facility.

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

2.8 Debugging

While developing a Cilk application, it is helpful to adopt a methodology to aid 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. 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. In this section, we offer a four-step methodological framework for debugging Cilk programs. This methodology includes a novel debugging tool, the Nondeterminator, which can greatly simplify the finding of "determinacy-race" bugs.

2.8.1 A debugging methodology for Cilk

The debugging methodology in this section is based mostly on tried-and-true strategies. In addition, we have great hopes for, but limited experience with, the Nondeterminator debugging tool. We hope that this four-step methodology will be helpful to you in developing bug-free Cilk code. Please let us know if you discover any other useful debugging techniques, or if you have problems with this methodology, by sending email to `cilk-developers@theory.lcs.mit.edu`.

Step 1. Get a serial Cilk program running.

A basic Cilk program differs from a C program by the addition of the parallel control constructs `spawn` and `sync`, which specify parallelism and synchronization. On one processor a Cilk program executes in a depth-first fashion like an ordinary C execution, and it conforms to the semantics of the C program that results when all `spawn` and `sync` keywords are removed. By including the file `cilk-sequential.h`—instead of `cilk.h`, `cilk-lib.h`, and `cilk-compat.h`—all Cilk keywords will automatically be removed from your Cilk program, allowing you to compile your Cilk code as a C program with an ordinary C compiler (for example, `gcc`). You can now run and test your serial Cilk program using ordinary C tools, such as a C debugger like `gdb`. Parallel debugging is harder than serial debugging, so try to get as many logic bugs out of your code as you can using this serial version before becoming embroiled in the vagaries of the parallel implementation.

Cilk provides a macro `Cilk_assert` to help you verify your Cilk program. The macro `Cilk_assert` is analogous to C's `assert`: if the assertion is false, a message is printed, the faulty process dumps core, and the program terminates. The postmortem dump can later be inspected using the `gdb` debugger. Use `Cilk_assert` to verify that the values computed at important points during your Cilk program satisfy program invariants.

`Cilk_assert` can be turned off by compiling your Cilk program with the preprocessor option `-DNDEBUG`, or with the preprocessor control statement `#define NDEBUG` in the beginning of your program. If `Cilk_assert` is turned off then these assertions induce no performance overhead during the running of your program.

Sometimes, it is helpful to use C code as an "executable specification" of the corresponding Cilk code. For example, you can use the Unix program `diff` to compare the outputs of your Cilk program and a corresponding C program to ensure that they do the same thing. Alternatively, you can use `Cilk_assert` to verify that the Cilk code is producing the same answers as the C code. The developers of the ⋆Socrates chess program used this strategy to great advantage. At each internal node of the game tree, the value of the game was computed redundantly by C code and compared with the value returned by the Cilk program. Although this strategy led to extremely slow code when assertions were turned on, it was possible to find many errors quickly, and the code ran perfectly fast when assertions were turned off.

Step 2. Use the Nondeterminator to eliminate determinacy-race bugs.

The next step is to verify that your program has no nondeterministic bugs. Most Cilk programs are ostensibly deterministic. That is, no matter how Cilk schedules the program, the results produced should be the same. Cilk provides a novel debugging tool, called the Nondeterminator, which can find bugs that cause nondeterminism.

If you are programming using nondeterministic features of C or Cilk, the Nondeterminator will not be as effective, since you have intentional nondeterminism in your code. Nondeterminism can arise in C when the behavior of your program is dependent on the results of random-number generators or time of day, which can change from run to run. Cilk's nondeterministic programming features, such as inlets, user locks, and abort, and the Cilk scheduler variable `SYNCHED` can also introduce nondeterminism. Usually, however, you can figure out some way of "turning off" this nondeterminism to produce an ostensibly deterministic version of your software. For example, in the ⋆Socrates chess program, a switch was included that could turn off the aspects of the program that produced nondeterministic behavior. Debugging a deterministic version of your code is much easier than debugging a nondeterministic version whether or not you use the Nondeterminator, but if you want to use the Nondeterminator, you should find a way to make your program deterministic.

Rewrite your deterministic Cilk program with headers as `cilk.h`, `cilk-lib.h`, and `cilk-compat.h` instead of `cilk-sequential.h`. Compile it for Nondeterminator execution using the sample `Makefile` included in the `nd-examples` directory of the Cilk distribution. Run the Nondeterminator version of your code on sample inputs to verify that your code is indeed deterministic. If your code contains nondeterminism, the Nondeterminator will locate the causes. Section 2.8.2 explains the operation of the Nondeterminator in more detail.

Step 3. Run your deterministic program in parallel.

Up to this point, all of your debugging has been on a serial machine. Compile your Cilk program with the ordinary Cilk compiler `cilk2c` and link it with the standard Cilk runtime library by using the sample `Makefile` in the `examples` directory. Run the program on your parallel computer by specifying the number of processors through the runtime option `-nproc`, as is described in Section 2.3. Unless you are unlucky (or sloppy, or too clever, or too stupid, etc.), your parallel Cilk code should run flawlessly, at least on input data sets that you tested with the Nondeterminator.

If your program exhibits bugs on a particular input, go back to Step 2 and run the Nondeterminator on that input. If the Nondeterminator does not localize the bug, it is likely that the bug is due to a violation of one of the Nondeterminator preconditions, such as the assumption that the serial program is deterministic.[8] Check the Nondeterminator guarantee given in Section 9.1 of the reference manual to see if a precondition is violated. You also may be able pick up the bug using `Cilk_assert`, as in Step 1. Debugging a dynamic computation is hard, however, and so at this stage, you are very much on your own in the wilderness. Most parallel programming environments do not have a tool like the Nondeterminator, however, and so you are still much better off than in many other parallel programming systems.

Step 4. Use assertions to verify the nondeterministic part of your program.

If your program is intended to be deterministic, this step is unnecessary. Otherwise, you have some nondeterminism that needs to be reintroduced for the final version of your program. Introduce the nondeterminism, and use `Cilk_assert` to verify that your code is running properly. If your nondeterministic code is buggy, there are few tools available for debugging beyond `Cilk_assert`. Segregating the nondeterministic from the deterministic part of your program, however, should greatly narrow your debugging focus.

2.8.2 Using the Nondeterminator

The Nondeterminator is a debugging tool to help users detect and locate ***determinacy races*** [13, 24] in their Cilk programs. A determinacy race is a cause of nondeterminism, and it arises when two parallel threads access (read or write) the same shared-memory location, and at least one of the accesses modifies the contents of the location. Assuming that the serial execution of a Cilk program is deterministic, the Nondeterminator either determines which locations in the program are subject to potential determinacy races when the program is run on the data set, or else certifies that the program is race free when run on the data set. The Nondeterminator is not a program verifier, because the Nondeterminator cannot certify that the program is

[8] Of course, the Cilk compiler and runtime system contain no bugs.

```
int x;

cilk void foo()
{
    int tmp;

    tmp = x + 1;
    x = tmp;
    return;
}
```

```
cilk int main()                     /* F */
{
    x = 0;                          /* e0 */
    spawn foo();                    /* F1 */
                                    /* e1 */
    spawn foo();                    /* F2 */
                                    /* e2 */
    sync;
    printf("x is %d\n", x);         /* e3 */
    return 0;
}
```

Figure 2.12: A simple Cilk program that contains determinacy races. The Cilk threads that make up the procedure `main()` are labeled in the comments at the right.

Case 1			**Case 2**		
F_1	F_2	e_3	F_1	F_2	e_3
`read x`			`read x`		
`write x`				`read x`	
	`read x`		`write x`		
	`write x`			`write x`	
		"`x is 2`"			"`x is 1`"

Figure 2.13: An illustration of a determinacy race in the code from Figure 2.12. The value of the shared variable `x` read and printed by thread e_3 can differ depending on how the instructions in the two instances F_1 and F_2 of the `foo()` procedure are scheduled.

race free for all input data sets. Rather, it is a debugging tool. The Nondeterminator only checks a program on a particular input data set. What it verifies is that every possible scheduling of the program produces the same behavior.

To illustrate how a determinacy race can occur, consider the simple Cilk program shown in Figure 2.12. The Cilk procedure `main()` spawns two procedures `foo()` which all read the shared variable `x` in the `tmp = x + 1` statement, and write `x` in the `x = tmp` statement. Since the operations in `foo()` are not atomic, both might read and write `x` at the same time. Figure 2.13 shows how this determinacy race can cause `x` to take on different values if the threads comprising the two instantiations of `foo()` are scheduled simultaneously.

The easiest way to use the Nondeterminator is through the standard `Makefile` included in the `nd-examples` directory of the Cilk distribution. The source code of the example from Figure 2.12 can be found under the name `race.cilk` in the `nd-examples` directory of the Cilk distribution. After installing the distribution, you can simply type:

```
> make race.nd
```

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

```
> race.nd
```

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

```
Determinacy race @ 0x30000: 'x'(race.cilk:10) with 'x'(race.cilk:9)
```

along with other output from the Cilk program and runtime system. The Nondeterminator finds a race in the program as we expect, because the write of `x` (whose address is `0x30000`) in one spawned procedure `foo()` conflicts with the read of `x` in another spawned procedure `foo()`. Besides the memory addresses 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 handles determinacy 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 `nd-examples` directory):

```
Free: Determinacy race @ 0xea8319e0: '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 determinacy 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 @ 0xea83195c: '(t+1)' (pllfree.cilk:47)
```

In order to produce easy-to-understand race messages, the Nondeterminator's memory allocation functions (see Section 6.13.1) do not recycle 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 much larger heap size than the ordinary version (or see Section 4.4).

If shared variables are updated by implicit inlets, the Nondeterminator will examine the types of these variables and whether the updates are commutative, that is, whether the same value results no matter what the order of updates. For example, the updates of an integer variable `x` in the following code are commutative:

```
x += spawn foo();
x -= spawn bar();
```

Since the updates are commutative and implicit inlets execute atomically, no race exists, and the Nondeterminator will not report one. If `x` is a floating-point number however, then the Nondeterminator will report race on `x`, because floating-point updates are not always commutative. For some codes, it is convenient to assume the floating-point updates are commutative; you can suppress this race report by using the runtime option `-float-comm` which tells the Nondeterminator to assume that floating-point updates are commutative.

Accesses on private variables (see Section 6.12.1) do not constitute races, because each thread has a logically separate instantiation of all private variables. Thus, private variables are not shared among threads.

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.

On average, the Nondeterminator version of the program exhibits a slowdown of around 15 to 20 times the serial execution time of the original optimized code. Even though the Nondeterminator detects determinacy races in parallel Cilk programs, the Nondeterminator itself, currently in this release, only runs on one processor. That is, you should not run the executable with `-nproc` n where n is not 1.

2.8.3 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 determinacy 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 determinacy race can occur.

As an example of handling Nondeterminator warnings, consider the program `queens-race.cilk` in the `nd-examples` directory which contains an n-queens puzzle program with determinacy races. The program uses the C library function memcpy for copying data from one location in memory to another. The memcpy library function has the prototype

```
void *memcpy(void *s1, const void *s2, size_t n);
```

If you run the Nondeterminator's cilk2c compiler on queens-race.cilk, among other warnings, you will see the following:

```
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
...
```

These warnings occur, because memcpy is not instrumented with for race detection.

We could ignore these warnings if we are sure that memcpy is not involved in a determinacy race, but that would be risky and would diminish the Nondeterminator's guarantee. (Moreover, the queens-race.cilk *does* contain a race!) Unfortunately, because memcpy is a C library function, we do not have access to its source code to run through the Nondeterminator's compiler.

To cope with this situation, the Cilk system provides a way for the user to notify the Nondeterminator about what shared memory is accessed by memcpy. The following two Nondeterminator library functions are provided:

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

Cilk_read_check informs the Nondeterminator that the shared-memory region of size bytes starting from p are read, while Cilk_write_check informs the Nondeterminator that the shared-memory region of size bytes starting from p are written.

In our example, line 48 of queens-race.cilk contains one of the offending calls to memcpy:

```
memcpy(new_config, config, i*sizeof(char));
```

We rewrite this line of code as follows (refer to queens-check.cilk in the directory nd-examples, which is the modified version of queens-race.cilk):

```
Cilk_read_check(config, i*sizeof(char));
Cilk_write_check(new_config, i*sizeof(char));
#pragma ndwarn -
memcpy(new_config, config, i*sizeof(char));
#pragma ndwarn +
```

The two `#pragma` statements turn off Nondeterminator warnings for the intervening code. Now, the Nondeterminator will properly check the call to `memcpy` and will no longer issue a warning. Of course, if the code contains many calls to `memcpy`, it may be productive to define a single inline function that provides the checks, turns off warnings, and calls `memcpy`, and use this inline function in place of `memcpy` in the code. None of the added statements introduces any overhead whatsoever during normal (non-Nondeterminator) execution.

Further documentation on using the `cilk2c` compiler for the Nondeterminator can be found in the Nondeterminator reference manual (Chapter 9).

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. This chapter describes some preliminary performance measurements taken of the example programs.

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-developers@theory.lcs.mit.edu`.

The example programs include the following:

- `blockedmul` — Blocked multiplication of two dense $n \times n$ matrices, written by Keith Randall.
- `notempmul` — A slightly less parallel, but more efficient, blocked multiplication of two dense $n \times n$ matrices, written by Keith Randall.
- `strassen` — Strassen's algorithm for multiplication of two dense $n \times n$ matrices, written by Michael Bender, Stuart Schechter, and Bin Song.
- `cilksort` — Sort a random permutation of n 32-bit integers, written by Matteo Frigo and Andrew Stark.
- `queens` — Backtrack search to solve the problem of placing n queens on an $n \times n$ chessboard so that no two queens attack each other, written by Keith Randall.
- `knapsack` — Solve the 0-1 knapsack problem on n items using branch and bound, written by Matteo Frigo.

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

Table 3.1: The performance of example Cilk programs. Times are in seconds. Measurements are for a complete run of the program, except for those programs that are starred (*), where because of large setup times, only the core algorithm was measured. Programs labeled by a dagger (†) are nondeterministic, and thus, the running time on one processor is not the same as the work performed by the computation. For these programs, the value for T_1 indicates the actual work of the computation, and not the running time on one processor.

- `lu` — LU-decomposition (without pivoting) of a dense $n \times n$ matrix, written by Robert D. Blumofe.
- `cholesky` — Cholesky factorization of a sparse symmetric positive-definite matrix represented as a quad-tree, written by Aske Plaat and Keith Randall.
- `heat` — Heat-diffusion calculation on an $m \times n$ mesh, written by Volker Strumpen.
- `fft` — Fast Fourier transformation of a vector of length n, written by Matteo Frigo.
- `Barnes-Hut` — Barnes-Hut n-body calculation, written by Keith Randall.
- `bucket` — Bucket sort of a random set of floating-point numbers, written by Guang-Ien Cheng.

Example programs can be compiled by typing `make <program-name>`. The makefile `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.

Table 3.1 shows speedup measurements that were taken of some of the example programs, as well as measurements of work T_1, critical path T_∞, and average parallelism $\overline{P} = T_1/T_\infty$. The machine used for the test runs was an otherwise idle Sun Enterprise 5000 SMP, with 8 167-megahertz UltraSPARC processors, 512 megabytes of main memory, 512 kilobytes of L2 cache, 16 kilobytes of instruction L1 cache, and 16 kilobytes of data L1 cache, running Solaris 2.5 and a version of Cilk-5 that used `gcc` 2.7.2 with optimization level `-O3`. The times measured are those of a complete run, except for `cilksort`, `lu`, `cholesky`, and `fft` (which are starred in the figure). For these codes, the setup time to read in the input was sufficiently long compared to the runtime that only the core algorithm was measured. For `cholesky`, numbers for two sparse matrices from the Harwell-Boeing test set [12] are reported. The matrix BCSSTK29 has dimension 13992, with 619488 nonzeros, or 0.3 percent of the matrix entries. The BCSSTK32 matrix has dimension 44609, with 1029655 nonzeros, or 0.05 percent of the entries. The two matrices were ordered using Matlab's minimum-degree ordering heuristic, which is not included in the `cholesky` benchmark. Regrettably, our time measurements are only accurate to within about 10 percent, due to unpredictability in today's deeply pipelined processor architectures, caused, for example, by direct-mapped caches and instruction alignment.

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

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

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

The slowdown of the Cholesky factorization is due the overhead in our quad-tree representation of sparse matrices.

The next few sections give a short description of each example program.

3.1 Fibonacci

The program is called `fib.cilk`.

Algorithm Description

This program contains probably the least efficient way to compute Fibonacci numbers [14]. 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.

Running the Program

Try `fib -nproc 8 24`. Other numbers for number of processors and the Fibonacci number to be computed are possible.

Typing `fib -help` lists the Cilk options that the runtime system accepts. Useful options include `-stats <n>` (to get more detailed statistics than just work, critical path, and average parallelism) and `-debug <n>` to print out debug information. If the program did not print out critical path and average 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 average parallelism may differ from run to run, especially if the number of processors is also changed.

Remarks

Fibonacci should give an almost perfect speedup: close to 8 on 8 processors, if reasonably large Fibonacci numbers are computed.

The file `fib-serial.c` contains the serial version of fib, for comparison.

3.2 Knary

Knary can be found in `knary.cilk`.

Algorithm Description

Knary is a synthetic benchmark. Its name really is k-n-ary. It can be used to generate arbitrary sizes of serial and parallel work. For more on Knary and its relation to the dag it generates, see Bobby Blumofe's thesis [3] (available from the Cilk homepage).

Running the Program

The command line is `knary -nproc <p> <serial> <parallel> <work> <depth>`. 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.

Remarks

Depending on the amount of parallel work and the amount of serialness that the command line parameters specify, different speedups can be achieved.

"Depth" is related to critical path. The way the (parallel) computation unfolds can be pictured as a graph. The depth of this graph is the critical path of the computation. The size of the graph is the computational work. The articles and theses on Cilk provide more detail on the theory of parallel computation. See the Cilk web page.

3.3 Sort

The file is called `cilksort.cilk`.

Algorithm Description

The file `cilksort.cilk` provides an implementation of a fast 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 $\Theta(\lg^3 n)$.

Running the Program

The command line is `cilksort -nproc p size` where `size` is a suitably large number of items to be sorted. Try one million.

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

Remarks

The implementation uses the above described algorithm for large arrays but falls back on a serial quicksort and mergesort for small arrays. The code includes an ordinary serial quicksort with some well-known optimizations [26].

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 an average parallelism of 300, suggesting that the implementation will continue to scale to larger numbers of processors.

The speedup measurements in Table 3.1 were taken with argument 4,100,000.

3.4 Matrix Multiply

Matrix multiply comes in three versions; all three are variations on a basic blocked matrix multiply. The differences lie in the way storage for temporary submatrices is handled.

The files are called `blockedmul.cilk`, `notempmul.cilk`, and `spacemul.cilk`.

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 multiply. 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.

These programs use a memory allocator, `Cilk_internal_malloc_fast`, which is faster than the standard `malloc`.

The serial algorithm used for the overhead column is not the naive triple for loop, since that performs abysmal due to the memory hierarchy of the test machine, but a blocked code like the divide and conquer Cilk algorithm. Note that for spacemul the

1 processor Cilk version runs slightly faster than the serial blocked code (an overhead factor of less than one), something that, in theory, should never happen, of course. This inconsistency is probably due to code-alignment differences.

Running the Programs

Assuming a matrix size of 1024×1024 and 8 processors, the programs can be started as `blockedmul -nproc 8 1024`, `notempmul -nproc 8 1024`, and `spacemul -nproc 8 1024`.

3.5 N-Queens

The file is called `queens.cilk`.

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 code is small and well documented. The parallelization strategy should be clear from the source code.

Note that the algorithm uses `abort` as soon as it has found a solution to stop the computation.

Running the Program

Try `queens -nproc 8 22`, for a board with 22 queens. (All measurements are taken with `queens 22`.)

Remarks

The runtime $T_8 = 18.8$ of `queens` in Table 3.1 exhibits a speedup of 9.3 over the runtime $T_S = 174.3$ of the serial C code. Since `queens` is a nondeterminstic program, it does not necessarily perform exactly the same computation as the C code. In the Cilk code, several possibilities for the placement of queens are examined in parallel. The serial C code examines these placements one by one. In contrast, the Cilk code examines them 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 superlinear 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.

3.6 Knapsack

The file is called `knapsack.cilk`.

Algorithm Description

Knapsack is a standard combinatorial optimization problem. As in queens, it uses speculative parallelism, but unlike queens, it does not use `abort`. The knapsack problem is to put the most valuable combination of items, each with a certain weight and a certain value, in a knapsack of limited capacity. 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. Note that the dependence on a stronger consistency model than dag consistency technically makes this a bad example program, since it violates one of the portability recommendations elsewhere in this manual.

Running the Program

Knapsack reads the list of items with weight and value from standard input. Two example files are provided as `knapsack.input` and `knapsack-run.input`. The program can be started as `knapsack -nproc 8 < knapsack.input`.

All measurements are taken with the input file `knapsack-run.input` (knapsack-run.input gives better speedups than knapsack.input. Try it.).

Remarks

Comparing runs for the two input files provides more insight into the reasons for super/sublinear speedup and speculative parallelism. The value of the heuristic bound, and the speed with which it is lowered, determine the difference between super and sub-linear speedup.

With a random input with only small variations in value/weight ratio, parallelism is not defeated by heuristic optimizations. The super-linear 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. In the case of small variations in the value/weight ratio (knapsack-run.input), 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 (knapsack.input), 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.

Unlike with queens, there is no easy way to get rid of the speculative parallelism by reformulating the problem.

3.7 Game

The file is called `game.cilk`.

As the problem size becomes small the spawns are replaced by ordinary C function calls, to reduce the Cilk overhead, trading off (unneeded) parallelism for serial speed. For programs with many small Cilk functions, this is a useful and often used optimization.

Algorithm Description

This is a simple game to test `aborts`. Rules: There are two players on a $n \times m$ board. Alternatively, each player puts a coin in position (i, j) and in all positions $(i\prime, j\prime)$ for all $i\prime \le i$ and $j\prime \le j$ (multiple coins count as one). The player who fills the board loses.

(Theorem: the first player has a winning strategy. Proof: homework.)

Running the Program

Simply `game`.

Remarks

This game was suggested by David Finberg, dfinberg@mit.edu.

3.8 Checkers

The file is called `ck.cilk`.

Algorithm Description

Again, a simple game to test `abort`. This program plays rudimentary checkers, without double jumps. You can't really play against it, but it does show some useful Cilk programming techniques.

Running the Program

The following options have both sides do a 5 ply search: `ck -nproc 8 -d 5`.
To have black do 5 ply and white 9 ply: `ck -nproc 8 -d 5 9`.

Remarks

This program is not interactive; you cannot play against it.

3.9 Dense LU Factorization

The file is called `lu.cilk`

Algorithm Description

LU performs the divide and conquer form of a blocked LU decomposition of a dense matrix ($A = LU$). 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.

Running the Program

Try `lu -nproc 8` or `lu -nproc 8 -n 128` for a matrix of size 128×128 instead of the default of 64×64. (All measurements in Table 3.1 are taken with `lu -n 2048`.)

Remarks

LU uses double precision. Matrices must be square, and dimensions must be a power of 2.

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] (available from the Cilk web pages).

This program only factors the matrix. It does not perform the subsequent forward and backward substition phase needed to solve $L\bar{y} = \bar{b}$ and $U\bar{x} = \bar{y}$.

3.10 Sparse Cholesky Factorization

The file is called `cholesky.cilk`

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.

Running the Program

A typical command line is `cholesky -nproc 8 < george-liu.mtx`. The option `-heap <size>` is needed when the matrix is large. The matrices used for Table 3.1, BCSSTK29 and BCSSTK32, are large.

All measurements are taken with matrices from the Harwell-Boeing set (see below). Command lines:

```
cholesky -nproc n -heap 400 < bcscol29.mtx
cholesky -nproc n -heap 400 < bcscol32.mtx
```

Remarks

Matrices can be found on the web on a site called Matrix Market (`http://math.nist.gov/MatrixMarket/`). Matrices BCSSTK29 and BCSSTK32 can be found in the Harwell-Boeing test set, as part of set BCSSTRUC5. The sparse Cholesky example understands the Matrix Market data format. In many sparse Matrix Market matrices, including the two used here, the actual floating-point numbers are not included. Only the nonzero structure is provided. The filter `randomize.c` in the example directory may be used to fill in the blanks. (Note, however, that it does not understand Matrix Market's two leading info lines.) The matrices are preordered using Matlab's column minimum degree ordering (symmetric minimum degree and Reverse Cuthill-McKee performed worse). BCSSTK29 has order 13,992 with 31,6740 nonzeroes before factorization and 2,299,310 after. BCSSTK32 has order 44,609 with 1,029,655 nonzeroes before factorization and 9,599,369 after.

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

During the (real) factorization of a positive-definite matrix, the diagonal elements remain positive. Violation of this condition is signaled by the algorithm taking the square root of a negative number, at which point the program aborts. In this case, the matrix is not positive definite, and a more general factorization method must be used, such as LU.

Like LU, Cholesky only factors the matrix. If the factorization is finished, it multiplies the factors again and checks whether the factorization was correct. The difference is printed and should be zero.

Like knapsack, the sparse code 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. The default block size of the program, which was used to generate the results in the table, is 4 by 4.

This sparse Cholesky code is not as sophisticated as other codes, such as the one found in the Splash-2 suite [27]. 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 retuned 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.

3.11 Heat Diffusion

The file is `heat.cilk`, and the accompanying parameter file is `heatparams`.

Algorithm Description

A Jacobi-type iteration can be used 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.

Running the Program

The program is invoked with `heat -nproc <p> <columns per partition>`, where `p` is the number of processors, and `<columns per partition>` are used to parameterize the granularity of the computation. Given the total number of columns, which is set in the parameter file `heatparams`, the number of partitions is the quotient of the command line paramter and the total number of columns.

Thus, to generate the results in the table, one would execute `heat 1`.

Experimental Results

Table 3.1 shows experiments with 4096 grid nodes in the i-dimension, 512 grid nodes in the j dimension, 1 column per partition, and 100 iteration steps (k-dimension). 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.

Remarks

The program was compiled with optimization `-O2`, and times are measured with critical path measurement on (which hardly impacts the run times for this application).

3.12 Fast Fourier Transform

The file is called `fft.cilk`.

Algorithm Description

This program is a highly optimized version of the classical Cooley-Tukey Fast Fourier Transform algorithm [10]. Some documentation can be found in the source code.

Running the Program

Type `fft -nproc 8 -c` for testing for correctness, and `fft -nproc 8 -s` for testing for speed.

Remarks

The default data size is 1024. Change the `#define SIZE` statement in the source code for different sizes. The program is optimized for an exact power of 2. 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.

Send comments and questions regarding this code to Matteo Frigo at athena@theory.lcs.mit.edu.

As can be seen in Table 3.1 on our test machine, we see the anomaly that the 1-processor Cilk version, which performs negligibly more work, is actually faster than the serial algorithm. This anomaly is due to unpredictability in today's deeply pipelined processor architectures, caused, for example, by direct-mapped caches and instruction alignment. We believe our time measurements to be accurate to within about 10 percent.

3.13 Magic Squares

The file is called `magic.cilk`.

Algorithm Description

This is a magic square:

8	1	6
3	5	7
4	9	2

The sum along all horizontal, vertical, and diagonal lines is the same.

The program computes the number of magic squares of size $n \times n$, in an original way.

Running the Program

The program is started as `magic -nproc 8 4` to have 8 processes compute that there are 4070 magic squares of size 4.

3.14 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, a single phase of the computation, the force calculations, is parallelized.[1] The force calculations are the bulk of the work of the algorithm and are also easy to parallelize, because the force on each particle can be calculated independently of the force on other particles.

Running the Program

`code -nproc` n `< sample-input`

`sample-input` runs the code for 2 force calcuations on 64K particles.

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`). This code is also the basis for the overhead comparison.

3.15 Bucket Sort

The file is called `bucket.cilk`.

Algorithm Description

Bucket sort is a linear time algorithm for sorting a set of 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

[1] We also have Cilk code in the directory `Barnes-Hut2` in which all four phases are parallelized. However, this code requires a stronger memory model than dag consistency and is therefore not very portable. This parallelization uses the `Cilk_lock` features of our SMP implementation — see Section 6.12 for details.

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 $\lfloor x/n \rfloor$'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.

Running the Program

Use the command line

`bucket -nproc` *nproc* *n*

where n is the size of the random array to sort. The program will print individual timing information for the three parallel phases—distribution of elements, the prefix-sum calculation, and placement of elements back into the array—and for the overall sort.

Part II — Implementation Notes

Chapter 4

Cilk Runtime System Implementor's Guide

Cilk-5.1 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-developers@theory.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-5.1 views memory

A Cilk-5.1 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-5.1 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-5.1 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-5.1 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. Six shadow spaces are needed altogether for the static region and for the heap region. The Nondeterminator algorithm uses one shadow space called ***reader*** for the read access, and the other called ***writer*** for the write access. Additional shadow spaces are needed to store information for both *reader* and *writer*: ***info*** stores the line number and file name of the variable, and ***var*** stores the variable names (used in the user's program) of the read or write access.

  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 four macros defined in `cilk-nd.h`, `STATIC_READER`, `STATIC_WRITER`, `HEAP_READER` and `HEAP_WRITER` tell how to get the *reader* and *writer* address from a shared memory address. In particular, they also determine the starting addresses of these four shadow spaces. The macro `SHADOW_INFO` defines how to get the address of *info* from a shadow space address, and the macro `SHADOW_VAR` defines how to get the address of *var* from a shadow space address.

- 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`.

- In order to give understandable race reports, The nondeterminator does not recycle memory. That is, when the user calls `Cilk_free`, the memory freed is not actually returned to the memory pool for later allocation. This feature can be disabled by changing the macro `WHEN_ND_NOALIAS` in `cilk-conf.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 (microscheduling). Cilk currently has no macroscheduler (the scheduling of multiple simultaneous Cilk programs), but we hope to provide a provably good macroscheduler 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

```
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 frames on the deque and the activation frames on the C stack.[1]

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

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

The procedure fib has one additional local variable, `_frame`, which holds a pointer to the procedure's frame. The frame is initialized and pushed onto the deque with 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

```
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`.

argument, and returns void. The first thing the microscheduled function does is to restore its state, first restoring the program counter using the `switch` statement, and then restoring local variables using the state-restoring code at each `sync` label.

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

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

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

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-5.1 scheduler will work for any reasonable consistency model supported by modern SMPs. The Cilk scheduler requires stronger consistency in the three following areas:

- The scheduler requires that obtaining a lock on a data structure guarantees atomic access to that data structure.
- When a frame is pushed on the deque, the scheduler requires that all writes must be globally visible before the tail pointer (the end of deque pointer) is incremented. In systems where writes can be reordered, a memory barrier of some sort must be inserted before the pointer increment. Writes remain ordered on our current platform (UltraSPARC SMP), so no barrier is needed for our implementation.
- When a frame is popped from the deque, the scheduler requires that the write of the tail pointer be globally visible before the read of the exception pointer is performed. In systems where reads can be performed before preceeding writes are globally visible, a memory barrier of some sort must be inserted before the read of the exception pointer. Our UltraSPARC SMP implementation requires a "StoreLoad" barrier at this point.

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

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 [16, Appendix B]). As compatibility of the C library with Cilk is a major problem, Section 6.13 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 [16]. 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]

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:

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

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

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

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

A `sync` statement may appear anywhere a statement is allowed in a Cilk procedure, including within 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* ($arg\text{-}decl_1$, ..., $arg\text{-}decl_n$) *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 $arg_2, \ldots, arg_n$. 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, $\ldots$, 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 $arg_2, \ldots, arg_n$ 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 Nonstandard features

Some releases of Cilk contain features that are not supported in all releases. These "nonstandard" features include private global variables, timers, and locks. 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 write a test program.

6.12.1 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 6.12.4).

6.12.2 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 2.3. 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 start, end;
 Cilk_time begin_cp, end_cp;
 Cilk_time begin_work, end_work;

 sync; /* hack to update timers */

 begin_cp = Cilk_user_critical_path;
 begin_work = Cilk_user_work;
 start = Cilk_get_wall_time();

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

 end = Cilk_get_wall_time();
 end_cp = Cilk_user_critical_path;
 end_work = Cilk_user_work;

 printf("cilksort took %f seconds.\n",
Cilk_wall_time_to_sec(end - start));
 printf("Work (of cilksort) = %f s\n",
Cilk_time_to_sec(end_work - begin_work));
 printf("Critical path (of cilksort) = %f s\n",
Cilk_time_to_sec(end_cp - begin_cp));
 printf("Average parallelism (of cilksort) = %f s\n",
Cilk_time_to_sec(end_work - begin_work) /
Cilk_time_to_sec(end_cp - begin_cp));
```

6.12.3 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 the standard features of Cilk. In a perfect world, we do not believe that application programmers should use protocols, and locking and unlocking a synchronization variable is a protocol. A language should perform protocols for the user. We hope to remedy this inadequacy of Cilk in the near future, but in the meantime, here is how to use Cilk's locks.

A lock has type `Cilk_lockvar`. The two operations 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 a pointer to an object of type `Cilk_lockvar`. Acquiring and releasing a Cilk lock has the memory consistency semantics of release consistency [17, p. 716]. The lock object must be initialized using `Cilk_lock_init()`.

Here is an example:

```
Cilk_lockvar mylock;
        :
        :
{
    Cilk_lock_init(&mylock);
        :
    Cilk_lock(&mylock);
        :
    Cilk_unlock(&mylock);
}
```

A simple example of code that uses locks is the bucket sort example in Section 3.15. A more complicated example can be found in the optimized version of the Barnes-Hut code, in Section 3.14.

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

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

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

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

If a program cannot be arranged such that locks are always acquired in some global order, deadlock avoidance becomes harder. In general, it is an NP-hard problem to decide whether a given program is deadlock-free. Even if the user can solve this NP-hard problem and 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 what is called an ***atomicity race***, where the counter update does not appear atomic because some access to it is unguarded. Currently, the Nondeterminator knows nothing about locks, so it cannot differentiate between a well-formed lock access and an access done without a lock. Thus, it must conservatively report a race on all accesses to the counter. Furthermore, locking can cause nondeterminism, which affects the guarantee that is provided by the Nondeterminator. See Section 9.1 for details. We are currently working on making the Nondeterminator lock-aware, so some of the hazards of using locks will mitigated by the use of the Nondeterminator.

6.12.4 OS-thread runtime system

Cilk-5.1can 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-developers@theory.lcs.mit.edu`.

6.13 Open issues

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

6.13.1 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`.

6.13.2 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 6.12.4) 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.

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 [22] 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 2.3).

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 6.13.1 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.

8.3 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 6.13.1 for a complete list of aliased functions.

Chapter 9

The Nondeterminator

The Nondeterminator is a debugging tool to help users detect and locate determinacy-race bugs in their Cilk programs [13]. Assuming that the serial execution of the program is deterministic, the Nondeterminator either determines which locations in the program are subject to potential determinacy races when the program is run on the data set, or else certifies that the program is race free when run on the data set. The Nondeterminator is not a program verifier, because the Nondeterminator cannot certify that the program is race free for all input data sets. Rather, it is a debugging tool. The Nondeterminator only checks a program on a particular input data set. What it verifies is that every possible scheduling of the program produces the same behavior. In this chapter we describe the functions and limitations of the Nondeterminator. We then summarize command-line options of `cilk2c` and of the Cilk runtime system that relate to the Nondeterminator. Finally, we explain the linguistic extensions of the Cilk language used by the Nondeterminator.

9.1 What the Nondeterminator guarantees

The Nondeterminator is a debugging tool to locate the determinacy-race bugs in Cilk programs. A ***determinacy race*** is a situation where two parallel Cilk threads access a shared-memory location, and one of the accesses is a write operation. The Nondeterminator is guaranteed to find determinacy races in a Cilk program run on a given input, or else to certify that no race exists, except in the following cases:

- *When the Cilk program run serially (on one processor) is already nondeterministic.* This situation may occur, for example, if the program has control structures guarded by a random-number generator, or if its behavior is dependent on the time of day. If a serial execution of a Cilk program is nondeterministic, then a determinacy race may still exist even if the Nondeterminator does not detect any races. But, if the Nondeterminator reports a race, the race is guaranteed to exist, although it may not be easily repeatable.

- *When different instantiations of inlets of the same procedure access the same variables.* In this case, determinacy races may exist on variables accessed by the inlets even though the Nondeterminator does not report any races. The Nondeterminator does not report these races, because it assumes, since programming with inlets is usually nondeterministic programming, that the user has already considered races among inlets. If the Nondeterminator reports a race between an inlet and a parallel thread, however, the race is guaranteed to exist. In addition, the Nondeterminator correctly detects races on variables accessed by implicit inlets (Section 2.7.1).

- *When the Cilk program uses* `Cilk_lock`. The use of locking is inherently nondeterministic. If several parallel threads attempt to acquire a lock, later control flow of the program may be affected by the order in which the threads gained that lock. The Nondeterminator currently does not understand anything about locking; it may report races between memory accesses which are protected by locking. The same scenario occurs when the program references the Cilk scheduler variable `SYNCHED`. We are working on a solution to these problems.

- *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`. In this case, determinacy races may still exist in these programs even if the Nondeterminator does not report any races. The user can manually instrument his code to specify how shared memory is accessed using the functions `Cilk_read_check` and `Cilk_write_check`, as is described in Section 9.4.

Note that the Nondeterminator is guaranteed to find at least *some* race if one exists. It is not guaranteed to find "all" races. Once there is one race, the control flow of the program after that point might be affected by the outcome of that race.

9.2 The Nondeterminator's `cilk2c` options

Besides the standard `cilk2c` options (Chapter 7), the Nondeterminator's `cilk2c` provides the following additional option:

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

9.3 The Nondeterminator's runtime options

Besides the standard Cilk runtime options (Section 2.3), a Cilk program linked with the Nondeterminator's runtime library accepts the following additional options:

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

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

`-nd-heap` *size* Set a maximum heap of *size* megabytes for use by the Nondeterminator. The default size is 16 megabytes.

9.4 The Nondeterminator's library functions

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.

9.5 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 `nd-examples` has an intentional determinacy 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, you need a recent version of `gcc` (we used `gcc-2.7.2`). Also, you will need the GNU linker installed (part of the binutils package, we used version 2.7), 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.

If you wish to edit the Cilk grammar, you will also need `bison` and `flex`.

How to obtain Cilk

Cilk can be obtained from our web page at `http://theory.lcs.mit.edu/~cilk` or directly by `ftp`ing the full distribution from

`ftp://theory.lcs.mit.edu:/pub/cilk/Cilk5.1.tar.gz`,

and the basic Cilk system without the Nondeterminator tool from

`ftp://theory.lcs.mit.edu:/pub/cilk/Cilk5.1.base.tar.gz`.

How to install Cilk

After copying `Cilk5.1[.base].tar.gz` into a directory on your system, perform the following steps:

- Type `zcat Cilk5.1[.base].tar.gz | tar xf -` , which will create a directory named `cilk` and put all of the Cilk software into this directory.

- Type `cd cilk` followed by `more README` . This file will list the installation instructions for Cilk-5.1. Follow the instructions to install Cilk, including the preprocessor (cilk2c) and the run-time system. The `README` file also includes instructions on running some test programs.

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 mail to

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

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

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

Reporting bugs

Please report bugs in the Cilk system by electronic mail (email) to

`bug-cilk@theory.lcs.mit.edu`

Appendix B

FAQ: Frequently Asked Questions

This chapter 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-developers@theory.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 `Makefiles` 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: 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 accessable 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-5.1:

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

We gratefully acknowledge the contributions of many others, including:

- Michael Halbherr (now of the Boston Consulting Group in Zurich, Switzerland) developed the PCM runtime system, the precursor of Cilk, and many of the design decisions in Cilk are owed to him.
- Arvind of MIT provided support and spiritual guidance throughout the Cilk project.
- Andy Shaw of MIT ported Cilk-1.0 to the Silicon Graphics Power Challenge SMP.
- John Litvin and Mike Stupak of Intel ported Cilk-1.2 to the Paragon running under OSF.
- Rolf Riesen of Sandia National Labs ported Cilk-1.2 to the Intel Paragon MPP running under SUNMOS.

- All those who have worked on previous versions of Cilk:

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

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

  Ching Law
  Igor Lyubashevskiy
  Adrian M. Soviani

Appendix D

Copyright and Disclaimers

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] 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.

[11] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, Santa Clara, California, April 1991.

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

[13] 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.

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

[15] 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.

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

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

[18] 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`.

[19] 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.

[20] 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`.

[21] 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.

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

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

[24] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[25] 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.

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

[27] 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