Cilk-4.1 (Beta 1) Reference Manual[1]

Supercomputing Technologies Group
MIT Laboratory for Computer Science

September 16, 1996

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

Contents

Chapter 1

Introduction

This document describes Cilk-4.1, 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. (See, for example, our world-class ⋆Socrates chess-playing program [11].) We have also had limited positive experiences on distributed-memory machines with numerical algorithms that exhibit good locality [4]. Unlike many other multithreaded programming systems, Cilk is algorithmic, in that the runtime system employs a scheduler that allows the performance of programs to be estimated accurately [5] based on abstract complexity measures.

1.1 What is included in Cilk-4.1?

This release includes the Cilk runtime system, the `cilk2c` compiler, a collection of example programs, and this manual. The Cilk runtime system included in this release runs on the *Sun Microsystems SPARC family of symmetric multiprocessors (SMP's)*. Earlier versions of Cilk have been ported to other platforms — including PVM, the Intel Paragon, the SGI Power Challenge SMP, and networks of workstations — but these ports are not available for this release. The current release works only on SMP's, but 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 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. 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 runtime system guarantees efficient and predictable performance.

Cilk grew out of work in both theory and implementation. The theoretical input to Cilk comes from a study of scheduling multithreaded computations, and especially of the performance of work-stealing, which provided a scheduling model that has since been the central theme of Cilk development. These results led to the development of a performance model that accurately predicts the efficiency of a Cilk program using two simple parameters: *work* and *critical-path length* [5, 7, 2]. More recent research has included *page faults* as a measure of locality [3, 4, 12].

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 [13] and which used work-stealing as a general scheduling policy to improve the load balance and locality of the computation [9]. 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 [2, 8].

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

Cilk-3 featured an implementation of dag-consistent distributed shared memory [4, 12]. With this addition of shared memory, Cilk could be applied 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.

With the current release Cilk-4, the authors of Cilk have changed their primary development platform from the CM-5 to the Sun Microsystems SPARC SMP. The compiler and runtime system have been completely reimplemented, eliminating continuation passing as the basis of the scheduler, and instead embedding scheduling decisions directly into the compiled code. The overhead in spawning of 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.

To date, prototype applications developed in Cilk include graphics rendering, protein folding [16], backtracking search, N-body simulation, and dense matrix computations. Our largest application is the ⋆Socrates chess program [11], which won second place out of a field of 24 systems 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. 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 [2, 12] contain fuller 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. Chapter 2 is a programmer's guide and introduces various features of the Cilk language. Chapter 3 is a reference for the Cilk language. The appendices provide a guide to install Cilk, a complete specification of the Cilk preprocessor and libraries, and miscellaneous documentation of this Cilk distribution.

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 describes the issues in programming with shared memory, and Section 2.6 shows how memory can be allocated. Section 2.7 describes some of Cilk's more advanced language features, including constructs for aborting already-spawned work. Tips for debugging and Cilk code are discussed in Section 2.8.

2.1 The Cilk extensions to ANSI C

One of the simplest examples of a Cilk program is a recursive program to compute the Fibonacci numbers. In C, this program might be written as shown in Figure 2.1(a). In comparison, Figure 2.1(b) shows a Cilk program that computes the nth Fibonacci number 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`. We anticipate that `cilk.h` will not be necessary in future releases, but, for now, it's mandatory — do not forget it.

The keyword `cilk` identifies a Cilk procedure. A procedure is the parallel equivalent of a C function. It may spawn other procedures in parallel and contain synchronization points. A procedure definition is identified by the keyword `cilk` and has an argument list and body just like a C function.

Parallelism is created when the invocation of a Cilk subprocedure is immediately preceded by the keyword `spawn`. Most of the work in a Cilk procedure is executed serially, just like C. 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 the child

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

returns, in the case of a Cilk spawn, the parent can continue to execute in parallel with the child. Indeed, the parallel-executing parent may 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 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 needed to support data-parallel programming. In Cilk, a `sync` waits only for the spawned children of a 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 not already present. As a consequence, a procedure never completes before all its children terminate.

The main procedure can be named either `main` or, for compatibility with existing Cilk-2 programs, `cilk_main`. The first form is preferred.

2.2 The Cilk model of multithreaded computation

To better understand Cilk's guarantees of efficiency, we now look at the model computation that Cilk supports.

A Cilk program, such as `fib` in Figure 2.1(b), consists of a collection of *procedures*, which are each broken down into multiple, atomic *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. (Arguments to these statements are considered part of the thread preceding the statement.) Thus, the instruction stream of a procedure is divided into threads by `spawn`, `sync`, and `return` statements. The first thread that executes when a procedure is called is the procedure's *initial thread*, and subsequent threads are *successor threads*. At runtime, procedures are structured as a rooted tree by the binary spawn relation, and the dependencies among their threads form a dag, as 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. For a program containing parallelism, however, the dependencies form only a partial order, permitting many ways of scheduling the threads in the dag. In actual implementation, 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

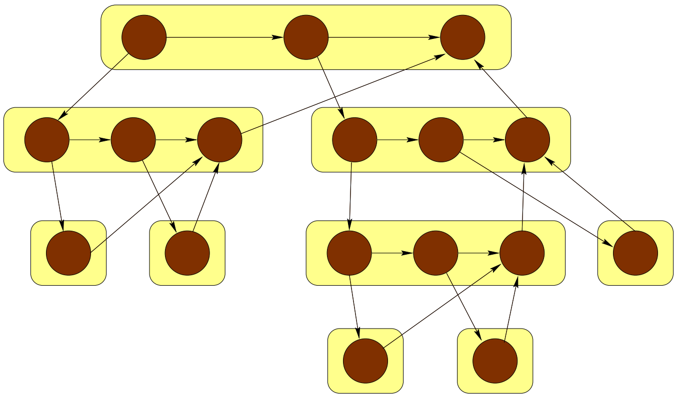

Figure 2.2: The Cilk model of multithreaded computation. Threads, shown as circles, are grouped into procedures, shown as shaded rectangles. Downward edges indicate procedure spawning, horizontal edges indicate continuation to successor threads, and upward edges indicate returning values to parent procedures. All three types of edges are dependencies which constrain the order in which threads may be executed.

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 dags, no good scheduling policy exists, in the sense that a dag can be constructed for which any schedule that provides linear speedup also requires vastly more than linear expansion of space [6]. Fortunately, every Cilk program generates a dag that can be scheduled efficiently, because the dag is *fully strict* [7], meaning that the only dependency edges that leave a subtree go from a thread in the root of the subtree to a thread in its parent. The dag generated by a Cilk program is guaranteed to be fully strict because the only dependencies leaving a subtree are generated by the `return` statements of the root procedure of the subtree.

The Cilk runtime system implements a scheduling policy based on randomized *work-stealing*. During the execution of a Cilk program, when a processor runs out of work, it asks another processor chosen at random for work to do. Locally, a processor executes procedures in ordinary serial order (just like C), executing the procedures 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 close to optimal time and space. From a theoretical perspective, there are two fundamental limits 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.[1] 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 close to 1 or 2 [5], and the formula

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

is a good approximation of runtime. This model predicts that a Cilk program with large average parallelism compared to the number of processors ($T_1/T_\infty \gg P$) will run with nearly linear speedup. Thus, the measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs.

We can bound the amount of space used by a parallel Cilk execution in terms of its serial space. (We assume here that space is stack allocated and not heap allocated as is provided by the C library function `malloc`.) Denote by S_P the space required for a P-processor execution. Then, S_1 is the space required for an execution on one processor. Cilk's scheduler guarantees that for a P-processor execution, we have $S_P \leq S_1 P$, which is to say one runtime stack per processor. In fact, much less space may be required for many algorithms (see [3]), 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, critical-path length, and space—together with the fact that you can count on them when designing your program—justifies Cilk as an *algorithmic* multithreaded language.

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.

[1] This theoretical model of execution time ignores cache effects, but is nonetheless quite accurate [5].

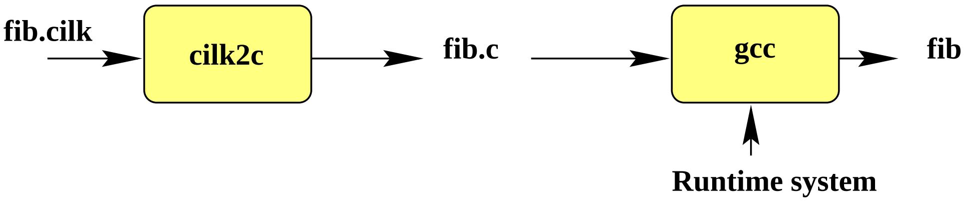

Figure 2.3: Compiling the Fibonacci program. Cilk code is passed through the `cilk2c` compiler, a type-checking preprocessor which generates C output. This output is then compiled with `gcc` and linked with the Cilk runtime library.

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,[2] you can simply type:

```
> make fib
```

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
Wall-clock running time on 4 processors: 2.655080
Total work = 9.529603
Critical path = 0.010000
Average parallelism = 952.960320
```

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. Note that the difference between the total work per processor and the overall execution time is due to the overhead of Cilk's scheduler. The third 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`. 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:

[2] 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 an `CILK_DIR` assignment that points to the distribution directory.

```
> fib -nproc 4 -stats 3 30
Wall-clock running time on 4 processors: 2.606318
Total work = 9.557002
Critical path = 0.010490
Average parallelism = 911.071400

RUNTIME SYSTEM STATISTICS:

 PN   Attempts     Steals     Faults    Threads
-----------------------------------------------
  0      28259         22         27    1675102
  1         70          4          6    1681378
  2      29164         16         23    1682819
  3      28790         12         19    1692045
-----------------------------------------------
TOT      86283         54         75    6731344


SUMMARY OF TIME SPENT BY EACH PROCESSOR:

 PN    Working      Total
-------------------------
  0   2.364104   2.460000
  1   2.370000   2.370000
  2   2.412898   2.504596
  3   2.410000   2.500000
-------------------------
TOT   9.557002   9.834596
```

Increasing the statistic level to 3 produces a table that reports the statistics of key runtime system operations in the Cilk runtime environment. Following that comes another table giving a the execution time spent on work and total time. The last line shows the sum of overall work time and total time.

These runtime statistics reveal some common patterns in the dynamic execution of Cilk programs. Of all the threads executed on each processor (listed under `Threads`), work-stealing migrates only a small number (`Steals`). The claim that work-stealing is an effective way of load-balancing among processors is reflected in the roughly equal work time (`Working`) on each processor, achieved with relatively few attempts to steal work (`Attempts`). The difference between the total time (`Total`) and the working time represents the amount of time spent doing work-stealing and other "nonproductive" runtime tasks. Little time is wasted by Cilk's scheduler in this example.

The critical path of the Fibonacci program is very short ($T_\infty \approx 0.01$ seconds compared to $T_1/P \approx 9.5/4 \approx 2.4$ 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 1, which prints the wall-clock running time, the total work and the critical path. Level 0 disables any printing of statistics. Some statistics are printed only if they are enabled during compilation by setting `CILK_CRITICAL_PATH = 1` in the `Makefile`. The highest meaningful statistics level is currently 4.

`-no-stats` Equivalent to `-stats 0`, that is, print no statistics.

`-debug` *mode* Enable diagnostic messages to be printed when the Cilk program is compiled with tracing enabled. The possible values of *mode* can be found in `cilk-conf.h`.

`-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* megabytes. The frame stack is used by `Cilk_alloca` to allocate stack storage.

`-bsize` *size* Set a maximum backing-store file size of *size* megabytes.

`-bfile` *filename* Set *filename* (which may include the directory path) of the backing-store file.

`--` 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`.

2.4 Cilk and C

Cilk does not have a real compiler, but only 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 3.15.1.

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()`. Including the file `cilk-compat.h` causes the standard names of Cilk-Unsafe library functions to be `#define`'ed to the Cilk version. Moreover, for certain Cilk-Safe functions that provide deficient semantics, such as `malloc()`, inclusion of `cilk-compat.h` also provides the original C versions under names that are preceded by `C_`, as in `C_malloc()`. Alternatively, you can `#define` just the ones you want by hand. See Section 3.15.1 for details.

2.5 Shared memory

When one Cilk procedure modifies a variable in parallel with another procedure's access to it, nondeterministic behavior can result. Consequently, it is important to understand Cilk's model for shared memory. At a minimum, any implementation of Cilk supports a coherence model called *dag consistency* [3, 4]. In addition, a Cilk implementation on particular machine architecture generally allows you to assume (at your own risk) the stronger shared-memory semantics of that machine.

Before delving into the semantics of shared memory, let us first see how sharing might occur in Cilk and what anomalies might arise. Figure 2.4 shows two Cilk procedures, `foo()` and `bar()`. In this example, `foo()` passes the variable `x` to `bar()` by reference, and then it proceeds to modify `x` itself before the `sync`.[3] Consequently,

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

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

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

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

Figure 2.4: An illustration of how shared access to variables can occur in Cilk.

it is not clear what values `bar()` will see when it reads `x` through the 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 what `foo()` will see.

Instead of worrying about this kind of problem, the easiest way to deal with anomalies of shared access is simply to avoid writing code like this. In other words, do not write programs in which a thread modifies a value that might be read or written in parallel by another thread. If you obey this rule, Cilk's dag-consistency model guarantees that your program will deterministically produce the same values for all variables, no matter how the threads are scheduled. Determinacy is a goal to be strived for in parallel programming, although sometimes more parallelism can be obtained with a nondeterministic algorithm.

In some circumstances you may find you need to cope with the intricacies of sharing. Cilk's dag-consistent shared memory provides you with a relaxed coherency model. The intuitive idea behind dag consistency is the following:[4]

> *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 coherency 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

[4]This definition is for intuition only, since it is not well defined for certain nondeterministic programs. See [3] for a proper definition of dag consistency.

consistency is that operations on shared memory are fast, both empirically and theoretically, even on multiprocessors with distributed memory.

In the meantime, if the machine on which you run Cilk supports a stronger coherency model, you are free to use it (at your own risk). Of course, your code will not be transportable 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.

As an example, on an SMP, you are free to use manufacturer-provided locking and unlocking primitives to update regions of memory atomically. For instance, some Unix systems provide a library header file `synch.h` which provides a variety of primitives for supporting mutual exclusion. Of course, you should be aware of the problems of deadlock that may occur, but if you are careful, Cilk will let you do what you want. As an aid to portability, Cilk provides a primitive `Cilk_fence()` that guarantees that all memory operations of a processor are committed before the next instruction is executed. Judicious use of `Cilk_fence()` can often yield faster code than 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* for stack-allocated storage, as is shown shown in Figure 2.5. From the point of view of a single Cilk procedure, a cactus stack behaves much like an ordinary stack. The procedure can allocate and free memory by incrementing and decrementing a stack pointer. The procedure views the stack as a linearly addressed space extending back from its own stack frame to the frame of its parent and continuing to more distant ancestors.

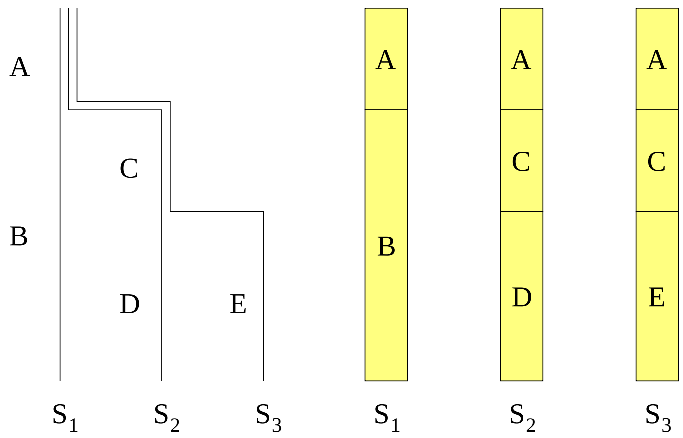

Figure 2.5: A cactus stack.

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 2.5. In the figure, subcomputation S_1 allocates some memory A and then spawns a subprocedure which allocates more memory B. Cilk's work-stealing scheduler allows another subcomputation S_2 to be started by stealing the oldest procedure belonging to S_1. The stolen procedure becomes the root of S_2, and a new branch of the cactus stack is started so that S_2's subsequent allocations do not interfere with the stack being used by S_1. The stacks as seen by S_1 and S_2 are independent below the steal point, but they are identical above the steal point. Similarly, when the oldest procedure of S_2 is stolen to start subcomputation S_3, the cactus stack branches again.

Cilk puts all local variables and arrays of a Cilk procedure into a frame on the cactus stack. In addition, you can allocate storage on the stack by calling

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

which allocates `size` bytes in the local stack frame and returns a pointer to the allocated memory. The memory is not cleared. When the Cilk procedure in which `alloca()` was called returns, the storage is automatically deallocated.

Cactus stacks have many of the same limitations as the ordinary C stacks [15]. 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 share storage that they create on the stack. 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.

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 be remedied soon. See Section 3.15.1.

2.6.2 Heap memory

Heap storage offers a way of alleviating some of the limitations imposed by stack 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. Cilk also provides a "penetrable barrier" mechanism, described in Section 2.7.3, which allows you to program your own scheduler within a procedure. Section 2.7.4 shows how a procedure can interact with Cilk's scheduler to test whether it is "synched" without actually executing a `sync`. Finally, Section 2.7.6 briefly discusses certain features for Cilk that are planned, but which did not make it into the current release.

2.7.1 Inlets

Ordinarily, when a spawned subprocedure 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.

An inlet is essentially a C function internal to a Cilk procedure. Normally in Cilk, the spawning of a subprocedure 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 subprocedure 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 2.6 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

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

`x` in the frame of the procedure that does the spawning. All the variables of `fib()` are available within `summer()`, since it is an internal function of `fib()`.

Since an inlet is like a C function, it obeys the restrictions of C functions (see Section 2.4) 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. Since it cannot contain `spawn`'s or `sync`'s, an inlet consists of a single 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, 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 without locking, declaring critical regions, or the like. On the other hand, if a thread not belonging to a procedure updates variables in the procedure's frame, Cilk guarantees nothing more than dag consistency in the interaction.

Inlets can take multiple arguments, but Cilk currently has the restriction that at most one argument to an inlet can be spawned and that this argument 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.

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

Implicit inlets are used by Cilk to support the accumulation of a result returned by a spawned procedure. Figure 2.7 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 instantly aborted, 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 2.8 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`

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

#define MAX_QUEENS 128

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[MAX_QUEENS];
    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;
    }

    /* copy the config from the caller */
    memcpy(new_config, config, i*sizeof(char));

    /* try each possible position for queen <i> */
    for (j=0; j<n; j++) {
        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;
    char config[MAX_QUEENS];
    char *result;
    int  i;

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

    n = atoi(argv[1]);
    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 2.8: 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 int bar()
    {
        abort;
    }

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

Figure 2.9: Illustration of an anomaly with the use of `abort`.

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 2.9.

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

2.7.3 Penetrable barriers

Cilk provides a "penetrable barrier" mechanism which allows you to program your own scheduler within a procedure. The idea of a penetrable barrier is that an inlet can use a `resume` statement to force the control of a procedure to proceed past a `sync` statement, even though there may be children of the procedure outstanding. This facility is especially important if, when within an inlet, you discover that you want to spawn a subprocedure. Cilk does not allow you to spawn a subprocedure within an inlet, but it does make it possible for the inlet to alter the main control of the inlet's procedure to allow the procedure itself to spawn the subprocedure.

The `resume` statement executed within an inlet normally has the same semantics as an ordinary `return` statement. When the inlet's procedure is stalled at a `sync`, however, it forces the control of the procedure past the `sync`, even though the procedure may have spawned children outstanding. Consequently, if an inlet wants the procedure to do something, such as spawn a subprocedure, the inlet can execute a `resume` instead of a `return`.

Figure 2.10 illustrates the use of the `resume` statement in a Cilk procedure `value` for minmax game-tree searching as might be used in a computer chess program. The procedure is part of a search heuristic called Jamboree search [11] that determines the value of a game position, stored in the search state `s`. The state `s` contains the board position, an interval $[\alpha, \beta]$ to search, a list `move_list` of possible moves, the number `num_children` of children in the move list, and the depth of the node in the game tree.

The state `ss` of the parent in the search tree is passed to the procedure `value`, along with the move to be made. After setting up its own board state `s`, the procedure first evaluates the first (0th) child recursively. Then, it evaluates the other children using the auxiliary procedure `test` (body not shown). The basic idea is that testing whether the subsequent children can beat the first child is much cheaper than establishing values for the subsequent children. Each child must be tested in turn, and if it fails the test, the child must then be valued. An array `status` keeps track of whether each child has been tested or valued. If the value returned by a search exceeds the upper bound `s.beta`, however, the entire search is pruned, since it can be proved that further search in this subtree is unnecessary.

The `while` loop iterates through the children, testing them, and if they fail the test, valuing them. When each child returns, it executes the `finished` inlet to set the child's status for the `while` loop. Within the inlet, if the child's value is greater than or equal to the bound `s.beta`, the other children are aborted, causing the `while` loop to terminate and the value of the search to be returned. Otherwise, the status of the search is updated.

The importance of `resume` can be observed in the situation when every child has been tested, but not all tests have completed. In this case, a `sync` is executed in the `while` loop to wait for the children to return. All is fine if the tests on the children succeed, since then no more work needs to be done on them. If one of the children fails its test, however, a value search must be performed on that child. When this

```
cilk int test(STATE *ss, MOVE move);

cilk int value(STATE *ss, MOVE move) {
  STATE s;
  int status[MAX_NUM_MOVES]; /* status of the subsearches */
  int next_child_to_test, next_child_to_value;
  int bestscore;             /* the bestscore kept so far */
  int aborted = 0;     /* the flag set when abort happens */

  inlet void finished(int child_score,
      int child_index,
      int test_finished)
  {
    bestscore = MAX(bestscore, child_score);
    if (bestscore >= s.beta) {
      aborted = 1;
      abort;
    }
    s.alpha = MAX(child_score, s.alpha);
    if ((child_score > test_alpha) && test_finished)
      status[child_index] = DO_VAL;
    else
      status[child_index] = DONE;
    resume;
  }

  /* Omitting code to create the new state "s" out of
     the parent's state "ss". This code makes the
     "move", tests the search depth (returning with
     a static evaluation of the board position if
     the depth is sufficiently deep), and generates
     the moves to be searched, placing them in the
     array "s.move_list". */

  best_score = spawn value(s, s.move_list[0])
  sync;

  if (bestscore >= s.beta) return (bestscore);
  s.alpha = bestscore;

  next_child_to_test = 1;
  next_child_to_value = 1;
  for (j = 1; j < s.num_children; j++) status[j] = DO_TEST;

  test_alpha = s.alpha;

  /* Spawn off the tests, and see if we need a value search */
  while (next_child_to_value < s.num_children && !aborted) {
    switch (status[next_child_to_value]) {
      case DO_VAL:
        status[next_child_to_value] = DOING_VAL;
        finished(spawn value(s, s.move_list[next_child_to_value]),
                 next_child_to_value,
                 0);
        break;
      case DO_TEST:
      case DOING_VAL:
        if (next_child_to_test < s.num_children) {
          finished(spawn test(s, s.move_list[next_child_to_test]),
                   next_child_to_test,
                   1);
          next_child_to_test++;
        }
        else
          sync;
        break;
      case DONE:
        next_child_to_value++;
        break;
    }
  }

  return(bestscore);
}
```

Figure 2.10: Illustration of penetrable barriers using the `resume` statement in minmax game-tree searching.

situation is discovered in the inlet, a `resume` forces the procedure past the `sync`, and the `while` loop continues, spawning the value search for the child. If, however, all children return from their tests without the need for value searches, the semantics of `sync` provide that control proceeds past the `sync` and the procedure terminates.

This elaborate example shows that programming with penetrable barriers can be tricky, and normally, you should not use this or other nondeterministic mechanisms unless your application demands it. Like aborts, penetrable barriers have subtleties of which you should be aware. One of the more tricky ones occurs in the common situation when an inlet desires its procedure to perform a spawn. If the procedure is not at a `sync` (for example, it has just executed a different spawn), it can be easy to overlook the fact that a spawn needs to be performed. It is essential to remember that an inlet can be interleaved in any matter whatsoever with the threads of a procedure, and you must be vigilant in ensuring that all possible interleavings yield correct behavior.

2.7.4 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. Cilk also allows a procedure to determine at runtime how many outstanding children it has.

The variable `SYNCHED` returns 0 if the procedure has outstanding children and 1 otherwise. 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`.

Another variable available is `SPAWN_COUNTER`, which contains the number of currently executing spawned child procedures. Naturally, if `SPAWN_COUNTER` is 0, then `SYNCHED` is 1. By inspecting `SPAWN_COUNTER`, a procedure can limit parallelism, which can be useful when the parallelism is speculative.

The variables `SYNCHED` and `SPAWN_COUNTER` are actually more like macros than honest-to-goodness variables. As will be explained in Section 2.8, the Cilk type-checking preprocessor 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` and `SPAWN_COUNTER` inspect Cilk's internal state. In the fast clone, which is almost always executed, the value of `SYNCHED` is the constant 1, and likewise, the value of `SPAWN_COUNTER` is 0. Thus, for the fast clone, `cilk2c` translates any Cilk code that tests these variables into C code that tests against constants. The optimizer in the C

```
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.

compiler recognizes that these tests can be performed at compile time, and thus the tests in the fast clone take zero runtime.

2.7.5 Shared and private variables

By default, all global variables defined in a Cilk program are shared among all threads. By shared, we mean that a global declaration of a variable `x` allocates a single memory location that all references to the variable `x` operate on. In particular, a write to variable `x` may eventually be seen by all threads in the program. (To understand exactly when a thread can be expected to see a write to `x`, see Section 2.5.)

Occasionally, however, it is useful to have greater control over access to a globally defined variable. The keyword `private` defines a global variable which has a separate instantiation for each thread in the execution (only logically - the implementation is actually very efficient). Private variables are useful for communicating between Cilk procedures and C functions. For example, in Figure 2.12, the private variable `option` is used to pass an option to a C function. Note that the two procedure invocations `fib(1)` and `fib(2)` can both use `option` to pass an option to their respective calls of `compute` without worry that the other invocation will overwrite their option setting.

Note that private variables are undefined at the start of each thread. In particular, private variables are not guaranteed to retain their values across `spawn`s, `sync`s, or procedure boundaries.

```
private int option;

void compute(void)
{
  ... compute something      ...
  ... read <option> variable ...
}

cilk void foo(int n)
{
   option = n;
   compute();
}

cilk int main(int argc, char *argv[])
{
   spawn foo(1);
   spawn foo(2);
   sync;
   return 0;
}
```

Figure 2.12: An illustration of the use of `private` variables.

2.7.6 Unimplemented features

Several features that we plan to put into Cilk did not make it into the current release. These 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.
- 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 and troubleshooting

Cilk does not have its own debugger. Instead, the user must use his native C debugger to debug Cilk code. Consequently, it is helpful to understand a few things about how Cilk is translated into C.

The Cilk type-checking preprocessor 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. The fast clone has the same name as the original Cilk procedure, and the slow version has the same name with `_slow_` appended. The fast clone is the one that usually executes. Only when Cilk's scheduler migrates a procedure to run on another processor does the slow clone execute, which Cilk's scheduler guarantees is not done often. In order to use a C debugger on Cilk code, you should look at the C output from `cilk2c`, which is what you must really debug.

In general, running a debugger on a Cilk program executing in parallel is a difficult, if not impossible, task. The main problem is the following: when a Cilk program starts, the runtime system performs some initializations, and then forks n processes, where n is the value specified by the `-nproc` option. These new processes in turn execute the real program in parallel, perform work-stealing, and so on, while the original process waits for the children to complete. Therefore, if you set a breakpoint on the main process, its children are not affected by the breakpoint and the program will never be interrupted. If you use `gdb`, one possibility is to attach the debugger to one of the child processes (`gdb` provides a special `attach` command to do that). You can only debug one process at the time, however, and you have no control over the activity of the runtime system.

The runtime system itself uses a debugging methodology that you might find useful. The code of the runtime system is crowded with calls to `CILK_ASSERT` and `Cilk_die`. The first is analogous to C's `assert`, in the sense that if the assertion is false, a message is printed, the faulty process dumps core and the all program terminates. In the same way, `Cilk_die` is analogous to `printf`, but it also dumps core and terminates the program. The post-mortem dump can later be inspected using `gdb`.

If you have a deterministic program, however, you can use a debugger like `gdb` to debug the program on one processor. Of course, the hard bugs will likely be ones where you think you ought to have a deterministic program, but it behaves nondeterministically.

Another strategy that works well for debugging is to have a serial C version of your program available as an "executable specification" of your Cilk program. In fact, you will often save lots of debugging time when developing a parallel application if you first develop a serial C version. Then, as you develop the Cilk version, you can use `CILK_ASSERT` to verify that the values computed at certain points during your Cilk program are identical to the corresponding values computed by your C program.

The developers of the ⋆Socrates chess program used this methodology 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. ⋆Socrates is actually a nondeterminate program, and so another strategy that was used was to have a switch that could turn off the aspects of the program that produced nondeterministic behavior.

The `PROBLEMS` file provided in the main Cilk directory discusses some of the common problems encountered when running Cilk and what, if anything, can be done about them. If you encounter and solve a problem that is not listed there, please send email to `cilk-developers@theory.lcs.mit.edu` so that your experience can be incorporated to help other Cilk users.

Chapter 3

Language Reference

This chapter defines the grammar for the Cilk language. The grammar rules are given as additions to the standard ISO C grammar (see [10, Appendix B]). Section 3.1 presents some fundamental concepts of the Cilk language, and the rest of the chapter introduces the syntax of all Cilk constructs. Compatibility of the C library with Cilk is a major problem. Section 3.15 discusses some of the issues associated with using the C library in Cilk programs.

The typographic conventions used in this chapter are the same of [10]. 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.

3.1 Basic concepts

In Cilk, the basic unit of scheduling and execution is a *thread*: a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (either explicit or implicit) statement. Threads are grouped together to form a Cilk *procedure*, which is the parallel equivalent of a C function. At runtime, procedure and thread spawning forms a dynamic tree, with data dependencies among threads forming a dag. (See Section 2.2.)

3.1.1 Frames

Each Cilk procedure has an *activation frame* storing all the local variables defined in the procedure. A frame is an area of memory shared by the initial and successor threads of the same procedure. Whenever a `spawn` occurs, the frame is pushed into a *frame stack* which is the runtime execution stack for Cilk procedures.

3.1.2 Runtime scheduling and work-stealing

As was discussed in section Section 2.2, the Cilk runtime system is designed to run multiple serial computations, one on each processor, in order to maintain the space bound of the equivalent of one execution stack per processor. In the meantime, it uses work-stealing to migrate computations on demand in order to balance the load among the processors.

At runtime, the Cilk scheduler manages one frame stack per processor that keeps procedure frames by their spawning depth. For local execution, the scheduler always takes the most recently spawned frame at the deepest level, according to the serial order of execution. The mechanism of work-stealing is implemented so that as soon as a processor's frame stack becomes empty, the processor attempts to steal a frame from a randomly chosen "victim." The frame chosen for the steal is the shallowest frame in the victim's frame stack. Intuitively, the shallowest frame is likely to contain more work, so the thief processor will have more work to do before it must steal again.

3.2 Program structure

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

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

3.3 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`, `resume`, `SYNCHED`, `SPAWN_COUNTER`, `shared`, `private`

The type qualifier `cilk` is used to declare Cilk procedures. Similarly, `inlet` is a type qualifier used to declare inlets (see Section 3.8).

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
```

3.4 Procedure definition and declaration

Cilk extends ANSI C with constructs to define procedures:

declaration: `cilk` type proc ($arg\text{-}decl_1$, ..., $arg\text{-}decl_n$) body

The keyword `cilk` specifies a 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 argument. Procedure bodies take the form of a C compound statement. Procedure bodies may also contain Cilk statements, including procedure `spawn`'s, `sync`'s, `inlet` declarations, and other Cilk constructs.

A procedure declaration can be used anywhere an ordinary C function declaration can be used.

3.5 Spawning

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

statement: var 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 *var* of *op* must be present if and only if *proc* has a non-void return type, and *var* 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, *var* 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.

3.6 Returning values

Within a Cilk procedure,

statement: `return` [expr]`;`

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

3.7 Synchronization

Within the body of a Cilk procedure,

statement: `sync;`

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

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

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

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

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

3.8 Inlet definition

An inlet is a short piece of code 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 `(` $arg\text{-}decl_1$, ..., $arg\text{-}decl_n$`)` 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 statements `abort`, `resume` and Cilk variables `SYNCHED` and `SPAWN_COUNTER` are allowed in inlet bodies. However, `spawn`, `sync`, and other `inlet` declarations are not allowed in inlet bodies. The scope of an inlet declaration is limited to the block in which it is declared.

3.9 Inlet invocation

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

statement: [var `=`] inlet `(`arg_1, ..., arg_n`)` `;`

where arg_1 should be a spawn argument as:

`spawn` proc `(`$proc\_arg_1$, ..., $proc\_arg_m$`)`

Arguments arg_2, ..., arg_n of the inlet and $proc\_arg_1$, ..., $proc\_arg_m$ of the procedure *proc* are first evaluated, and *proc* is then invoked as in the `spawn` statement above. Control of the calling procedure may resume at the statement after the inlet invocation in parallel with the execution of *proc*. When *proc* completes execution, *inlet* is invoked with the result returned from the spawned procedure, if any, and with its other arguments arg_2, ..., arg_n. Note that *inlet* runs in the lexical context of the procedure defining *inlet*, and thus it can reference automatic variables in its parent procedure.

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

3.10 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.

3.11 Resuming

Within the body of a Cilk inlet,

statement: `resume` [expr]`;`

can be used to end the inlet and resume execution of the procedure. The semantics of `resume` is the same as that of `return`, except that `resume` causes the procedure to resume execution if it is suspended at a `sync`, penetrating the synchronization barrier without waiting for all outstanding child procedures to complete. This penetrable barrier mechanism can allow the procedure to gain additional control over the execution. For example, the procedure may continue to spawn more procedures or change variables to affect the execution of other spawned procedures.

3.12 Cilk scheduler variables

The read-only Cilk scheduler variables `SYNCHED` and `SPAWN_COUNTER` allow 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 there are no outstanding child procedures, and 0 otherwise.

The scheduler variable SPAWN_COUNTER returns the number of outstanding child procedures. The variable SYNCHED can be defined in terms of SPAWN_COUNTER:

SYNCHED ≡ (SPAWN_COUNTER == 0)

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

3.13 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 procedures, or even other threads of the same 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.

3.14 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 [`=` init];

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.

The keyword `shared` is used to explicity designate a variable as shared.

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. This default is necessary to eliminate bad behaviors exhibited by the C library when its variables are shared. We hope to rectify this ungraceful situation in a future release.

Warning: Local static variables (static variables defined inside Cilk procedures or C functions) are private variables! For reasons we can't control (i.e. `gcc` won't let us), we cannot make local static variables shared by default. We do not recommend programming in Cilk with local static variables.

3.15 Open issues

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

3.15.1 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`.

3.15.2 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.

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 4

Copyright and Disclaimers

Chapter 5

Installation Instructions

Currently, these installation procedures only apply to a Solaris SMP machine. Hopefully, they will not need much modification for installing on other systems.

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.

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

How to obtain Cilk

Cilk can be obtained by `ftp`ing the file `Cilk4.1.tar.gz` from
`theory.lcs.mit.edu:/pub/cilk/Cilk4.1.tar.gz`

How to install Cilk

After copying `Cilk4.1.tar.gz` into a directory on your system, perform the following steps:

- Type `zcat Cilk4.1.tar.gz | tar xf -` , which will create a directory named `Cilk-4` and put all of the Cilk software into this directory.

- Type `cd Cilk-4` followed by `more README` . This file will list the installation instructions for Cilk-4.1. 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

Chapter 6

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 ⟨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;
```

Internals

See [14] 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 7

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 7.1, and the library functions in Section 7.2.

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 7.3.

7.1 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.

7.2 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.

7.2.1 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 3.15.1 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.

7.2.2 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()`.

7.2.3 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);
```

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

7.3 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 3.15.1 for a complete list of aliased functions.

Chapter 8

Cilk Runtime System Implementor's Guide

There is no Implementor's Guide yet. If you wish to port Cilk to an unsupported platform, please contact `cilk-developers@theory.lcs.mit.edu`.

Chapter 9

Notes on Compiling Cilk

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 macro scheduler (the scheduling of multiple simultaneous Cilk programs) but we hope to provide a provably good macroscheduler in a future release.

9.1 Nanoscheduling

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

The nanoscheduler is very easy to implement. The `cilk2c` compiler translates each Cilk procedure into a C procedure with the same arguments and return value. Each spawn is translated into its equivalent C function call. Each sync is translated into a NOP, because all children have completed by each sync when scheduling in this fashion.

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

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. When a spawn occurs, the state of the current procedure is saved in this frame and the frame is pushed on the bottom of the frame deque. When the spawn completes, the frame of the spawning procedure is popped, and the procedure continues its execution.

Figure 9.1 shows the nanoscheduled version of the Fibonacci example from Figure 2.1(b). `cilk2c` generates a frame for the `fib` procedure to hold all of its state. The state of `fib` includes its argument `n` and its two local variables, `x` and `y`. Additional state, such as the procedure ID and program counter, are stored in the header.

The procedure fib has one additional local variable, `_frame`, which holds a pointer to the procedure's frame. The frame is initialized 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. It then pushes the frame on the frame deque with the macro `_PUSH_FRAME`. The spawned procedure is then called with a normal C function call. After the function call returns, the frame is popped and 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`.

Finally, the nanoscheduler checks 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 has been migrated, `_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.

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.

9.2 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.

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

```
struct _fib_frame {
StackFrameT 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)
      return (n);
    else
      {
        int x;int y;
        { _frame->header.entry=1;
          _frame->scope0.n=n;
          _PUSH_FRAME(_frame);
          x=fib(n-1);
          _XPOP_FRAME_RESULT(_frame,0,x);
        }
        { _frame->header.entry=2;
          _frame->scope1.x=x;
          _PUSH_FRAME(_frame);
          y=fib(n-2);
          _XPOP_FRAME_RESULT(_frame,0,y);
        }
        /* sync */;
        return (x+y);
      }
  }
}
```

Figure 9.1: Nanoscheduled version of `fib`.

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

For each return in the Cilk code, instead of returning the value as in C, the slow version uses `_SET_RESULT` and `_SET_VALID` 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 put in in the appropriate final destination.

At each synchronization point, the slow version needs to 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 saves its state (entry 3 in the code), marks the frame as suspended using the `_HEAPIFY` macro, and returns.

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.

9.3 Overheads

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

- A procedure's frame needs to be initialized at the start of each procedure. This operation requires only a few writes.

- A procedure's state needs to be saved before each spawn. The entry number and each live, dirty variable must be saved.

- A procedure's frame needs to be pushed before each spawn and popped after each spawn. These are just pointer increments and decrements and can be implemented with only one read and one write. Note that some memory synchronization operations are needed for architectures which do not implement sequential consistency (see Section 9.4 for details).

```
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));_SET_VALID;return;}
    else
      {
        { int _temp0;
          _frame->header.entry=1;
          _frame->scope0.n=n;
          _PUSH_FRAME(_frame);
          _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;
          _PUSH_FRAME(_frame);
          _temp1=fib(n-2);
          _XPOP_FRAME_RESULT(_frame,/* return nothing */,_temp1);
          _frame->scope1.y=_temp1;
          if (0) { _sync2:; }
        }
        { if (_SYNC) {
            _frame->header.entry=3;
            _HEAPIFY(_frame);
            return;
            _sync3:;
          }
        }
        {_SET_RESULT((_frame->scope1.x+_frame->scope1.y));
         _SET_VALID;return;}
      }
  }
}
```

Figure 9.2: Microscheduled version of `fib`.

- 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 2.5 times slower than the sequential C program on one processor. For programs with larger thread lengths, this overhead will be minimal.

9.4 Interactions with weak memory models

In general, the Cilk-4.1 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 head 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 head 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.

Chapter 10

Release Notes

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.

Chapter 11

People

The Cilk-4.1 development team consisted of the following people:

Robert D. Blumofe
Mingdong Feng
Matteo Frigo
Charles E. Leiserson
Aske Plaat
Keith H. Randall
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:

  Bradley C. Kuszmaul
  Christopher F. Joerg

Phil Lisiecki
Howard Lu
Robert Miller
Richard Tauriello
Daricha Techopitayakul
Yuli Zhou

Bibliography

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

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

[3] Robert D. Blumofe, Matteo Frigo, 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.

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

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[6] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 362–371, San Diego, California, May 1993.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[8] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International*

Symposium on High Performance Distributed Computing (HPDC), pages 96–105, San Francisco, California, August 1994.

[9] 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.

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

[11] 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`.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.