Cilk 3.0 (Alpha 1) Reference Manual[1]

October 18, 1995

[1]Cilk is trademark of the Massachusetts Institute of Technology. The Cilk project has been partially supported by ARPA contracts N00014-94-1-0985 and N00014-92-J-1310. Computer time on the CM5 was provided by the Scout project under ARPA contract MDA972-92-J-1032.

Contents

Chapter 1

Introduction

This document describes Cilk 3.0, an ANSI C-based language and runtime system for multithreaded programming. Cilk is designed for computations with dynamic, highly asynchronous, tree-like parallelism, which are typically difficult to write in data-parallel or message-passing style. Divide-and-conquer algorithms and backtracking search are examples of tree-like computations that are particularly well-suited to Cilk.

1.1 What is Included in Cilk 3.0

This release includes the Cilk runtime system, the Cilk-to-C preprocessor (cilk2c), a collection of example programs, and this manual.

The Cilk runtime system included in this release runs on two multiprocessor platforms: the Thinking Machines CM-5 and the Sun Sparc SMP. It also runs on uniprocessors running Unix or Linux.

Cilk has been ported to other platforms not included in this release: PVM, the Intel Paragon, the SGI Power Challenge SMP, and networks of workstations. For more information about these implementations, or for help with porting Cilk to a new platform, contact the Cilk group by sending electronic mail to `cilk-developers@theory.lcs.mit.edu`.

1.2 Background and Goals

Cilk is designed to support dynamic, tree-like parallelism on message-passing architectures. The philosophy behind Cilk is that a programmer should concentrate on structuring a program to expose parallelism, while the runtime system dynamically schedules the computation to run efficiently on any number of processors. The runtime system takes care of details like load balancing and communication protocols.

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

[1] This chapter was written by Yuli Zhou and Rob Miller.

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 [BJK+95, BL94].

The first implementation of Cilk was a direct descendent of PCM/Threaded-C, a C-based package that provided continuation-passing-style threads on the CM-5 and used work-stealing as a general scheduling policy to improve the load balance and locality of the computation [HZJ94a, HZJ94b].

The next release, Cilk 2.0, showed significant improvements. The language is now fully typechecked, supports all of ANSI C in its C-language subset, and offers a simpler syntax for writing multithreaded procedures. The runtime system has been made more portable, and the base release includes support for several architectures other than the CM-5. The provably-efficient scheduler has been incorporated into the runtime system, and many performance improvements have been made.

The current release, Cilk 3.0, includes our implementation of dag-consistent distributed shared memory. With this addition, we are able to solve a much wider class of applications. Dag-consistency is a very weak but nonetheless useful consistency model, and its relaxed semantics allows for an efficient, low overhead, software implementation.

A key principle in the Cilk development methodology is that the runtime mechanisms and language features are closely tied to the development of practical applications. To date, major applications developed in Cilk include graphics rendering, protein folding [PJGT], backtracking search, N-body simulation, and the *Socrates chess program [JK94], which won 3rd place in the 1994 ACM ICCC (International Computer Chess Championship), and recently won 2nd place in the 1995 ICCC.

1.3 The Cilk Model of Multithreaded Computation

A Cilk program consists of a collection of "procedures," where each procedure is broken down into multiple "threads." For example, here is a Cilk procedure that computes the nth Fibonacci number:

```
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);
    }
}
```

In the body of the procedure, the `spawn` statements invoke recursive calls to `fib` via the scheduler, which may execute them in parallel and possibly on different processors. The `sync` statement suspends a procedure until all of its spawned child procedures return, so that their return values can be used afterwards.

In Cilk terminology, a *thread* is a sequence of non-blocking code within a procedure. A procedure is divided into threads by the `sync` statements. The thread that starts the procedure is called the *initial thread*, and threads that start after a `sync` are called *successor threads*. At runtime, the spawn relationships between procedures form a tree, and the data dependencies among their threads form a dag. This is illustrated in Figure 1.1.

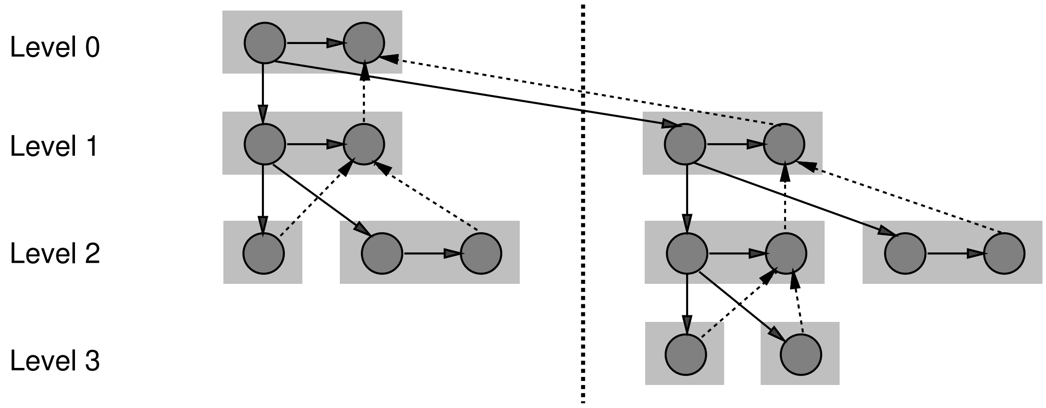

Figure 1.1: The Cilk model of multithreaded computation. Threads, shown as circles, are grouped into procedures, shown as shaded rectangles. Downward solid edges indicate procedure spawning, horizontal solid edges indicate continuation to successor threads, and upward dashed edges indicate returning values to parent procedures. All three types of edges are data dependencies which constrain the order in which threads may be executed.

A correct execution of a Cilk program must obey all the data dependencies in the dag, since a thread cannot be executed until all the threads it depends on have completed. For a program containing parallelism, however, the data 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. Every active procedure has associated state that takes up storage, and every data dependency between threads assigned to different processors requires communication. Thus, different schedules may have drastically different effects on the space and time efficiency of 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 providing linear speedup requires more than linear expansion of space [BL94]. The problem can be solved by restricting the class of dags to those generated by *strict* computations, where a spawned procedure only sends values to its own ancestors in the spawning tree. A subclass of strict computations, called *fully strict* computations, is especially interesting in that a procedure only sends values to its parent. Since this reflects exactly the ordinary procedure call-return semantics, all dags formed by Cilk procedures are fully strict.

The Cilk runtime system implements a general scheduling policy based on *work-stealing*. In work-stealing, the work of a computation is distributed dynamically among all the processors involved in the computation. Locally, a processor executes procedures in sequential order, working depth-first through the dag. Spawned procedures are placed in a ready queue until the scheduler can execute them. When a processor has no work to do, it steals a ready procedure from another processor chosen at random. Thus, work-stealing balances the load in distributed, probabilistic fashion.

Cilk's work-stealing scheduler executes a fully strict computation in time and space close to optimal. From a theoretical perspective, there are two fundamental limits to how fast a Cilk program can run. The first limit is the amount of *work* in the program, or the total time needed to execute all threads in the dag, which we equate with the program's uniprocessor execution time and denote by T_1. The second limit is the program's *critical path*, or the time needed to execute threads on the longest path of data dependency, which we equate with the program's execution time on an infinite number of processors and denote by T_∞. Cilk's work-stealing scheduler executes a fully strict computation on p processors in time $O(T_1/p + T_\infty)$, which is optimal. In practice the constant factor has been measured very close to 1 [BJK+95]. Thus, for programs with plenty of parallelism ($T_1/T_\infty \gg p$), the Cilk scheduler provides linear or nearly linear speedup.

From an implementation perspective, we all know that the sequential execution (i.e., depth-first schedule) of the dag is both space efficient and has good locality. The Cilk scheduler is simply running multiple sequential computations, one on each processor. Assuming S_1 to be the space required for the sequential execution on one processor, a p-processor execution thus requires no more than $S_1 p$ space, or the equivalent of one runtime stack per processor.

To this end, the Cilk runtime system sorts the ready procedures on a processor in order of their level in the spawning tree (see Figure 1.1). Locally, a processor executes procedures in depth-first order, always executing a procedure from the deepest level of the spawn tree. When another processor requests work, however, the processor gives it a ready procedure from the *shallowest* level of the tree, since procedures at shallower levels represent more work. In addition, the last returning child that enables its parent procedure migrates the parent to its own processor for execution, in keeping with the "one stack per processor" invariant.

1.4 Ongoing and Future Work

The next release of Cilk is expected to include more sophisticated synchronization and control mechanisms for Cilk procedures. With *inlets*, a procedure will be able to specify some work that is performed as soon as a spawned child returns, such as spawning off additional children, or even prematurely terminating the remaining children. This mechanism will enable Cilk procedures to perform various kinds of speculative computation which in the current Cilk release can only be achieved by explicit continuation-passing (threads).

Another project that will soon come to fruition is Cilk-NOW, a portable system for scheduling Cilk programs on a network of idle workstations [BP94]. Cilk-NOW is dynamic, fault-tolerant, and adaptive. Workstations can enter and leave the parallel computation at any time without disruption, and a Cilk job's share of the available workstations changes dynamically with the job's changing level of parallelism. The initial release of Cilk-NOW is expected soon.

1.5 About This Manual

This manual is primarily intended for users who wish to write Cilk application programs. It also serves as documentation for implementors who wish to port Cilk to other platforms. Chapter 2 is a programmer's guide and introduces various features of the Cilk language and runtime system interface. Chapter 3 discusses the shared memory features which are new to Cilk 3.0. Chapter 4 is a reference for the Cilk language. The appendices provides a complete specification of the Cilk runtime system, a guide to porting the Cilk runtime system, and some miscellaneous documentation of this Cilk distribution.

Chapter 2

Programming in Cilk

This chapter introduces the Cilk language from a programmer's perspective. It shows how to write, compile, and run a simple Cilk program, then describes in more detail the language features that a beginning Cilk programmer needs to understand. We assume that the reader is already familiar with C.

Figure 2.1 shows the structure of the Cilk system (for the CM-5). An application programmer writes a Cilk program, which is transformed to ordinary C code by the Cilk preprocessor `cilk2c`, and then compiled and linked with the runtime system using the C compiler on the target machine. During execution, the runtime system uses native message-passing facilities on the parallel machine to schedule, communicate, and synchronize the threads in the program. On the CM-5, for example, the runtime system uses Strata, a basic message-passing library that supports efficient active messages. Cilk has also been implemented on shared-memory architectures by simulating message-passing.

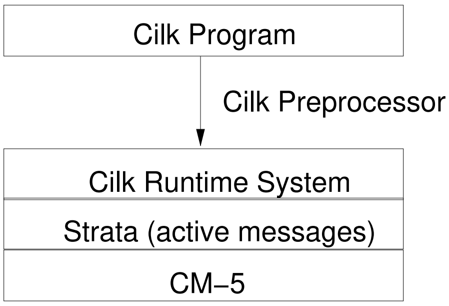

Figure 2.1: Overall structure of the Cilk package on the Thinking Machines CM-5

[1] This chapter was written by Yuli Zhou and Rob Miller.

2.1 Getting Started

The first code we are going to write in Cilk is a program that computes the Fibonacci function recursively. In C, this program could be written as shown in Figure 2.2.

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

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

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

        return (x+y);
    }
}

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

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

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

Figure 2.2: Sequential C program that computes the Fibonacci function.

In comparison, Figure 2.3 shows a Cilk program that computes the Fibonacci function. Notice how similar the two programs look. In fact, the only differences between them are an extra include file (`cilk.h`) and a few keywords: `cilk`, `spawn`, `sync` and `cilk_main`. The `cilk` keyword identifies a Cilk procedure. Within a procedure, other procedures can be `spawn`'d to create parallel threads of execution, and the `sync` statement waits for all spawned procedures to complete so that their return values can be used. The procedure `cilk_main` is the first procedure executed in a Cilk program. These keywords and others will be described in more detail in section 2.2. But first, we will compile and run this simple Cilk program.

```
#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 cilk_main (int argc, char *argv[])
{
    int n, result;

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

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

Figure 2.3: Cilk program that computes the Fibonacci function.

2.1.1 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 the CM-5. We assume some basic knowledge of Unix and C program development on the CM-5.

Cilk program files, which end with `.cilk` by convention, are first transformed to ordinary C code by the Cilk preprocessor `cilk2c`, producing `.c` files. The resulting C program files are compiled using the C compiler and linked with the Cilk runtime library. The easiest way to get started is to use the standard makefile included in the `examples` directory of the Cilk distribution, which automates all of these steps.

The source code for Fibonacci in Cilk may be found under the name `fib.cilk` in the `examples` directory of the Cilk distribution. After installing the distribution[2], you can simply type:

```
cm5> make fib_cm5_st
```

to make the executable. The `_cm5` extension is used to denote executables that run on the CM-5, and the `_st` extension is for executables with statistics gathering enabled. To run the program, type

```
cm5> jrun fib.job
```

to execute `fib_cm5_st` on the CM-5, using the DJM job script `fib.job` to request an appropriate machine partition.

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

```
+ fib_cm5_st -nproc 32 30
fib (30) running on 32 processors
Result: 832040
32 processors participating, execution time 2.384271 seconds
PE     Local/ Stub Migratd Clos  Queue   Heap  Lev Requests
-------------------------------------------------------------
 0:   126557/  41     119   64     32    887   29     4979
31:   126495/  86     134   29     13    418   29      523
...
PE      Work    Sched  Polling     Term    Other    Total  Broken
-------------------------------------------------------------------
 0:   1.3012   0.8347   0.2844   0.0007   0.0049   2.4258       1
21:   1.2720   0.8672   0.2819   0.0012   0.0032   2.4256       1
...
Critical path: 0.015099 seconds
```

[2]If you are using `scout` at MIT, on which the Cilk distribution is already installed, it is only necessary to make a copy of the `examples` directory. Make sure that the `STRATA_DIR` and `CILK_DIR` assignments in the `Makefile` point to each of their distribution directories respectively (see Chapter 6.1 for the locations of these directories).

After the output of the program itself ("Result: 832040"), 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. Next is a table that reports the numbers and sizes of key data structures in the Cilk runtime environment. Following that comes another table giving a breakdown of the overall execution time spent on various activities, plus an indicator (`Broken`) of the accuracy of the timing statistics.[3] The last line reports the critical path length of the program.

These numbers reveal some common patterns in the dynamic execution of Cilk programs. Of all the threads executed on each processor (`Local`), work-stealing migrates only a small number of them (`Migratd`). The number of closures (i.e., activation frames of procedures) that are allocated at any time (`Clos`) is fairly small, as is the size of the ready queue (`Queue`), reflecting the moderate space requirements due to Cilk's "one sequential computation per processor" model. (Note that we invoked the `fib` procedure with an argument of 30, so a sequential computation would use a stack at most 30 activation records deep. The `Clos` and `Queue` statistics are within a small constant factor of the sequential space requirements.)

The claim that work-stealing is an effective way of load-balancing among processors is reflected in the roughly equal work time (`Work`) on each processor, achieved with relatively few requests to steal work (`Requests`). Since the threads in the `fib` procedure are quite short, the scheduler overhead (`Sched`) is significant compared to work time, but the overhead-to-work ratio would be very small with sufficiently long threads.

The critical path of the Fibonacci program is very short (0.015 seconds compared to 2.38 seconds of 32 processor execution time). The Cilk performance model (see section 1.3) thus predicts near linear speedup over one-processor execution time. To verify if this is really the case, you can modify the `fib.job` script to run the program on various numbers of processors, specified by the `-nproc` option.

2.2 Writing Simple Cilk Programs

A Cilk program is made up of Cilk procedures and ordinary C functions. As we described in section 1.3, a procedure is further broken down to threads, which are the basic units of scheduling. Actually, the Cilk language also supports the abstraction of threads (see section 2.4), which can be used to access the capabilities of the runtime system directly. However, for simple programming tasks the procedure abstraction should prove to be both sufficient and much easier to work with.

2.2.1 Procedures

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. The following is the `fib` procedure shown previously:

[3] The `Broken` columns counts the number of "broken" threads – threads which are interrupted by a context switch on the CM-5.

```
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);
    }
}
```

The actual work in a Cilk procedure is performed by ordinary C code, which is executed sequentially. Parallelism is created when a procedure is spawned. Spawning a procedure creates an activation frame for the procedure, which is subsequently managed by the scheduler. The programmer specifies neither where nor exactly when the procedure will be executed. It is up to the scheduler to execute it at some later time, and possibly on a different processor.

A procedure cannot use the return values of the procedures it has spawned until it executes a `sync` statement to wait for all of them to return. The `sync` statement is the point where the current thread of the procedure terminates, and the scheduler suspends the current procedure and performs other work. When all of the child procedures return, execution of the procedure resumes at the point right after the `sync` statement, which starts a successor thread.

Spawns and syncs expose the parallelism in a program, without specifying how the parallel threads should be scheduled. This separation of parallelism from actual scheduling decisions is critical to Cilk, whose runtime system depends on work-stealing to dynamically migrate procedures among processors for load balancing. We have already explained in section 1.3 why such a generic scheduling policy works well for fully strict programs, which at this point includes all Cilk programs written in procedure style.

In practice, work-stealing works best with "bushy" spawning trees, where each procedure spawns only a few children. Bushy spawning trees are better because each steal carries a large amount of work, which minimizes the number of procedures migrated by work-stealing. In contrast, a procedure which spawns hundreds of children in a tight loop, where each child has only a small amount of work to do, executes poorly in the work-stealing model. The same program could be restructured in a more tree-like manner, so that the original procedure spawns two procedures, which in turn spawn two procedures, and so on until the leaves of the computation are reached. The restructured program should perform much better under work-stealing.

2.2.2 Sequential Calls

Every Cilk procedure has a "sequential version," an equivalent C function which can be executed with the special keyword `call`. The sequential version is derived from the procedure by removing all `sync` statements and changing all `spawn`s to `call`s. Calling a procedure sequentially has much lower overhead than spawning it in parallel (by about an order of magnitude), so this feature is useful when the procedure body is expected to execute quickly. For example, one may wish to call `fib` sequentially once the argument drops below 10:

```
cilk int fib (int n)
{
    if (n < 2) return(n);
    else if (n < 10) return (call fib (n-1) + call fib (n-2));
    else
    {   int x, y;

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

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

When a procedure is called sequentially, all computation within that procedure is also executed sequentially, including all of its child procedures and their descendents. Thus, every sequential call reduces the amount of parallelism in the Cilk program, so these calls should be used with care. Sequential calls are very useful for testing and debugging, however, since a sequential computation is completely deterministic. The standard Cilk option `-serial` causes the entire Cilk program to run serially by calling the sequential version of `cilk_main`.

2.2.3 The First Cilk Procedure: `cilk_main`

The procedure `cilk_main` is treated specially by the Cilk preprocessor, which generates initialization code that makes it the first procedure executed by the scheduler. Like the C `main` function, `cilk_main` has the prototype

`cilk int cilk_main (int` *argc*`, char *`*argv*`[]);`

and receives the command line arguments as its parameters. For example, here is the definition for `cilk_main` in the Fibonacci program:

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

    n = atoi(argv[1]);
```

```
    result = spawn fib(n);

    sync;
    printf ("Result: %d\n", result);

    return 0;
}
```

A Cilk program can have a C `main` function instead of `cilk_main`, but then the user is responsible for initializing the Cilk runtime system and starting the scheduler. For more information, see section 2.5.3.

2.2.4 Standard Runtime System Options

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* is 0, use all processors in the partition. *n* must be at least 0 and no greater than the partition size. The default is 0, using all processors in the partition.

`-serial` Run in serial mode. `cilk_main` is called sequentially, so all spawns in procedures descended from it are likewise turned into sequential calls.

`-print-stats, -no-print-stats, -no-stats` Turn on/off statistics printing at the end of program execution. `print-stats` is the default. The Cilk program must be compiled with statistics enabled for these options to be meaningful.

`-debug` *mode* Enable diagnostic messages to be printed when the Cilk program is compiled with tracing enabled.

`-no-good, -no-provably-good` These options turn off a feature of the work-stealing scheduler which is necessary in theory for guaranteeing efficient time and space bounds, but is sometimes costly in practice.[4]

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 `cilk_main`.

For example, one can invoke the `fib_cm5_st` program by

[4] By default, the last returning child of a procedure always migrates its parent to its own processor. This behavior is necessary in theory to guarantee the provably efficient time and space bounds. In practice, however, this extra migration costs some Cilk programs up to 10% of their running time on the CM-5, and even more on SMP implementations of Cilk. The `-no-provably-good` option (or its synonym, `-no-good`) eliminates the extra migration.

```
cm5> fib_cm5_st -nproc 8 30
```

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

2.3 Passing Arrays to Procedures

Note: most of this section is obsolete with the introduction of shared memory in Cilk 3.0. We include this section for completeness, as all of the array passing features described in it are still valid, but using shared memory to pass arrays to procedures is the preferred method.

In a C program, there are two ways of sharing data structures between function invocations: by passing pointers to data allocated on the stack or in the heap, and by using external (global) variables. Neither of these techniques works in Cilk, because of work-stealing. First, pointer arguments that were valid on the processor where a procedure was spawned become meaningless when the procedure is stolen to a different processor. Second, "global" variables in a Cilk program are not actually global. Every processor has a private copy of the program's global variables, making these variables useless for communication and shared storage.

This section introduces a Cilk extension that addresses the problem of sharing data structures without resorting to globally coherent shared memory[5]: passing arrays by value. Pass-by-value array arguments turn out to be quite useful, which we illustrate by writing a Cilk program to solve the n-queens puzzle.

Given a (hypothetical) chess board of size `n` by `n`, our problem is to find out the number of ways to place `n` queens on the board such that none of them attack each other (i.e., no two queens are on the same row, column, or diagonal). The Cilk program is shown below.

```
#define MAXQUEENS 20

int safe (int path[], int row, int col);

cilk int search (byvalue int path[MAXQUEENS], int n, int i)
{
    if (i >= n) return 1;
    else
    {
        int ncount[MAXQUEENS];
        int j, sum = 0;

        for (j=0; j<n; j++)
          if (safe (path, i, j)) {
              path[i] = j;
              ncount[j] = spawn search (path[..], n, i+1);
```

[5] As of Cilk 3.0, Cilkhas dag-consistent global shared memory that can be used for sharing data structures between procedures. See Chapter 3 for details.

```
            }
            else ncount[j] = 0;

          sync;
          for (j=0; j<n; j++) sum += ncount[j];
          return sum;
      }
  }
```

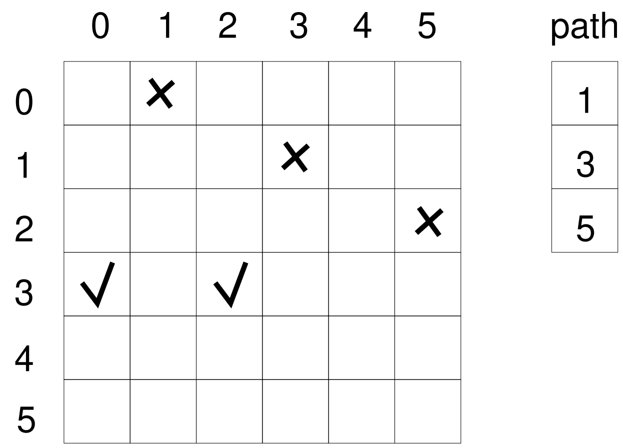

Figure 2.4: Diagram showing one search at row 3, for $n = 6$. A cross marks the position of a queen already placed, and checks show where recursive searches are spawned. The `path` vector is passed as initial argument to the current search, which records positions of queens that are already placed on the board.

Our solution is organized as a search by the rows on the board. As illustrated in Figure 2.4, the array argument `path` is used to record the positions of queens that have already been committed to rows 0 to `i-1`. These positions are used to enumerate the safe positions for the queen on row `i`, and for each such position, a sub-search is spawned in parallel.

In the argument list of `search`:

```
cilk int search (byvalue int path[MAXQUEENS], int n, int i)
```

the declaration of `path` declares a pass-by-value array argument of size `MAXQUEENS`. The actual array is passed in the spawn statement:

```
ncount[j] = spawn search (path[..], n, i+1);
```

where `path[..]` specifies the array object that is copied into the frame. In Cilk, the expression a`[`l`..`u`]` specifies a section of an array from a`[`l`]` to a`[`u`]`. Since `path` is declared to have fixed size, the two bounds in the section are not specified, and are assumed to be 0 and `MAXQUEENS` $- 1$.

In this program we would like to save some space by making the size of the `path` vector adjust to the depth of the search. Such a variable sized array argument can be declared by omitting the array size:

```
cilk int search (byvalue int path[], int n, int i)
```

Since the size of the array may now vary from one invocation of the procedure to another, it must be specified when the procedure is spawned. One way to do this would be to use the array section syntax introduced previously, copying the new `path` from the current `path`. Unfortunately, this won't help after we switch to variable-length `path` arrays, since the current `path` only contains `i` elements, and the new path must contain `i+1`. Instead we shall use an *initialization pointer*, a mechanism that allows the programmer to allocate the array in the frame, then initialize it explicitly:

```
int *new_path;
ncount[j] = spawn search (:new_path[i+1], n, i+1) {
                memcpy ((char*)new_path, (char*)path, i*sizeof(int));
                new_path[i] = j;
            };
```

In the above code, `new_path` is the initialization pointer. The expression `:new_path[i+1]` allocates space for an integer array of size `i+1`, and sets `new_path` to point to the beginning of the array. The spawn statement itself has an optional trailing statement, which here is used to initialize the array. The trailing statement is executed as part of the code to spawn the procedure.

Note that the same technique can be used to have `ncount` allocated in the frame as a variable size array, eliminating the need for the `MAXQUEENS` limit on the size of the board.

2.4 Threads and Explicit Continuation-Passing

To keep its implementation simple, the Cilk runtime system only understands threads and schedules them as the basic units of execution. In this sense, Cilk procedures exist as linguistic abstractions which are translated into threads by the Cilk preprocessor. In order to access the capabilities of the runtime system more directly, the thread abstraction is also supported as a subset of the Cilk language. The following example shows the Fibonacci computation written as two threads:

```
thread fib_T1 (cont int k, int x, int y);  /* forward declaration */

thread fib_T0 (cont int k, int n)
{
    if (n < 2) send_argument (k, n);
    else
    {   cont int k1, k2;

        spawn_next fib_T1 (k, ?k1, ?k2);
        spawn fib_T0 (k1, n-1);
        spawn fib_T0 (k2, n-2);
    }
```

```
}

thread fib_T1 (cont int k, int x, int y) {
    send_argument (k, x+y);
}
```

In the thread style, the successor thread in the `fib` procedure is split off as a separate thread `fib_T1`, leaving the initial thread as `fib_T0`.

A thread is spawned by making a *closure*. A closure is a data structure containing a pointer to the thread's code, plus all the arguments needed to execute the thread. In Cilk, the keyword `spawn` is used to create the initial thread of a child procedure, and the keyword `spawn_next` is used to create a successor thread in the current procedure. While child threads are typically created with all arguments available, successor threads usually need to wait for some arguments to arrive in the future. To enable such communication, the sender of the argument must first obtain a reference, called a *continuation*, to the empty slot in the waiting closure. Figure 2.5 shows the closures created after executing `fib_T0` and the continuations passed among child and successor threads. This mechanism, called *explicit continuation-passing*, is used to implement the procedure call/return linkage of Cilk.

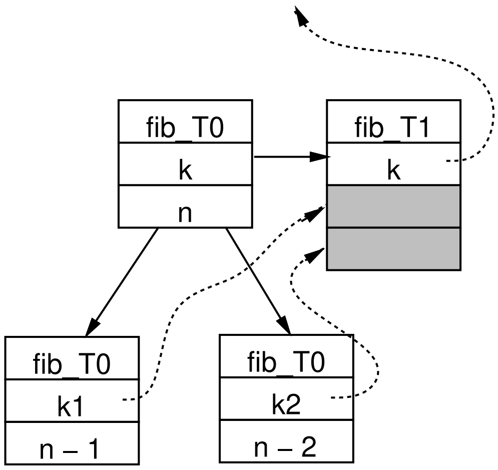

Figure 2.5: Closures created during execution of fib threads

Continuation variables are typed. The type of a continuation is formed from the type of the argument slot to which it points, with the type qualifier `cont` indicating that it is a continuation. In the `fib` example, `k1` and `k2` are declared as continuations pointing to integer arguments:

```
cont int k1, k2;
```

Continuation variables are initialized when the successor thread is spawned. For instance, in the code

```
spawn_next fib_T1 (k, ?k1, ?k2);
```

the form ?k1 specifies an empty argument slot in the closure of fib_T1, and stores in k1 a continuation referring to the slot. A thread that is passed a continuation k can send an argument to the waiting closure using the statement

send_argument (k, *arg*);

The type of *arg* must be coercible to the base type of the continuation to fit the size and semantics of the slot in the closure.

Threads were the only parallel programming extension supported in release 1.x of Cilk. Release 2.0 introduced procedures, which raise the level of abstraction of Cilk programming. Most Cilk programs can and should be written entirely with procedures. Cilk threads survived in release 2.0 because they reflect directly the scheduling mechanisms implemented in the runtime system, and because certain features needed by Cilk system programming and some user applications only exist in threads. Those features — accumulators, signals, tail calls, and migration threads — are described in the following sections. First, however, we look at how the threads and procedures in your Cilk program can interact and spawn each other.

2.4.1 Spawning Threads from a Procedure, and Vice Versa

All spawns made by a Cilk procedure use *implicit continuation passing*, meaning that instead of specifying a continuation where the result should be sent, the spawn specifies a local variable that should receive the result. As a consequence, when a procedure spawns a thread, it omits the thread's initial argument, the continuation. The continuation is supplied automatically by the Cilk preprocessor, using a reference to the local variable.[6]

For instance, suppose we wanted the fib procedure to do half its work using the fib_T0 thread introduced in the last section. The resulting code is shown below.

```
thread fib_T0 (cont int k, int n);  /* thread declaration */

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

        x = spawn fib_T0 (n-1);
        y = spawn fib (n-2);

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

[6]Actually, the local variable should be omitted when the spawned procedure or thread returns void. Even in this case, however, the Cilk preprocessor supplies an implicit continuation to the spawned procedure or thread.

```
}
```

The procedure spawns `fib_T0` in precisely the same way as it recursively spawns itself, by assigning the spawn to a local variable. Note that it passes only one argument to `fib_T0`, not two, since the continuation argument is handled implicitly.

Now consider the other direction: spawning a procedure from inside a thread. All spawns made by a Cilk thread use *explicit continuation passing*, meaning that a continuation should always be passed as the first argument. For a spawn of a Cilk procedure, this explicit continuation does not correspond to any of the procedure's formal arguments, but the Cilk preprocessor will automatically treat it as the procedure's return continuation. Thus, when the Cilk procedure returns, it will send its return value to the explicitly specified continuation.

Thus, here is a version of `fib_T0` which uses the Cilk procedure `fib` for half of its work:

```
cilk int fib (int n);  /* procedure declaration */

thread fib_T0 (cont int k, int n)
{
    if (n < 2) send_argument (k, n);
    else
    {   cont int k1, k2;

        spawn_next fib_T1 (k, ?k1, ?k2);
        spawn fib (k1, n-1);
        spawn fib_T0 (k2, n-2);
    }
}
```

2.4.2 Accumulators

Accumulators are the first feature we will consider that is supported only by Cilk threads, not by Cilk procedures. Accumulators can be used to gather results from several threads into a single argument slot. To use an argument slot as an accumulator, the following syntax is used to specify the initial value (*init*) and the number of values to be accumulated (n) at thread spawning time:

`spawn_next` *thr* `(`. . . , `?`*k*{*n*:*init*}, . . . `);`

The continuation thus captured can be used to accumulate values with the statement

`accumulate (`*k, op, val*`);`

where *op* specifies the operation used to accumulate values, which can be one of `add`, `mul`, `and`, `or`, `min`, `max`, or a user defined function (see section 4.7.3). Exactly n values should be sent to activate the waiting thread, but the actual accumulation can happen in any order.

The following shows the Fibonacci example using an accumulator:

```
thread fib_T1 (cont int k, int x);  /* forward declaration */
```

```
thread fib_T0 (cont int k, int n)
{
    if (n < 2) accumulate (k, add, n);
    else
    {   cont int k1;

        spawn_next fib_T1 (k, ?k1{2:0});
        spawn fib_T0 (k1, n-1);
        spawn fib_T0 (k1, n-2);
    }
}

thread fib_T1 (cont int k, int x)
{
  accumulate (k, add, x);
}
```

2.4.3 Signals

Signals are accumulators without any values. The syntax for creating a signal slot is the same as for an accumulator slot except that there is no initial value:

`spawn_next` *thr* `(`$\ldots$, ?$k\{n\}$, $\ldots$`);`

The continuation k must have type `cont signal`. Also, in the prototype for *thr*, the argument slot for the signal must be of type `signal`. To send a signal to a continuation, use

`send_argument (`k`);`

A signal is useful for indicating that a thread has completed without having to communicate a value. The following shows an example in which we use signals to create a barrier in the code. The thread `spawner` runs several `task` threads and its successor, `barrier`. Each `task` signals the `barrier` thread when it has done its work. The `barrier` thread sends a value to continuation `k` when all of the signals have arrived.

```
thread barrier (cont int k, signal) { send_argument (k, ...); }

thread task (cont signal s, int i)
{
    ... do i'th piece of work ...
    send_argument (s); /* signal completion */
}

thread spawner (cont int k)
{
```

```
    cont signal s;
    int i;

    spawn_next barrier (k, ?s{10});
    for(i = 0; i < 10; i++) spawn task (s, i);
}
```

2.4.4 Direct and Tail Calling Threads

Threads can also be called as C functions. Syntactically, this just involves changing the keyword from `spawn` to `call`, as shown in the following example:

```
thread fib_T0 (cont int k, int n)
{
    if (n < 2) accumulate (k, add, n);
    else
    {   cont int k1;

        spawn_next fib_sum (k, ?k1{2:0});
        spawn fib_T0 (k1, n-1);
        call fib_T0 (k1, n-2);
    }
}
```

A direct call of a thread is essentially the tail call optimization frequently used in implementing functional languages. Unlike sequential execution of Cilk procedures, a directly called thread may still spawn threads and leave them to execute in parallel. This feature is thus strictly an optimization that avoids making a closure and the subsequent scheduling overhead. Since the scheduler needs some work to do after the current thread finishes, direct calling a thread as the last action, whenever possible, generally makes the computation more efficient.

Cilk also provides the keywords `call_next`, `tail_call` and `tail_call_next`, where the `_next` suffix is used for calling successor threads. A tail call can only be issued as the last action in a thread to call the thread itself. It is implemented as a jump to the beginning of the thread after resetting the arguments, thus is even more efficient than `call`, which involves C function call overhead.

2.4.5 Pass-by-value Array Arguments to Cilk Threads

Array arguments as described in section 2.3 can be passed to Cilk procedures and Cilk threads alike. Specific to Cilk threads, however, one can create an array of empty slots in a waiting closure, and send elements of the array individually.

The following example declares an array argument called `result`, which is used to collect data from other threads for the `combine` thread:

```
thread combine (cont int k, byvalue int result[3]);
```

When spawning the combine thread, we can use the following syntax to capture a continuation k1 that refers to the first element in the array. References to other elements in the array are formed by performing arithmetic on the continuation:

```
spawn_next combine (k, ?k1[]);
spawn child1 (k1, ...);
spawn child2 (k1+1, ...);
spawn child3 (k1+2, ...);
```

Continuation arithmetic is analogous to pointer arithmetic, and the same operators are supported. See section 4.7.4 for more details.

2.4.6 Migration Threads

The Cilk runtime system contains a simple storage allocator (see section 2.5.4) which can be used to create data structures shared among threads executing on the same processor. The Cilk language provides a protocol through which the programmer can arrange for the data to be migrated together with the thread.

The protocol only works with Cilk threads, not Cilk procedures, and requires the programmer to define two threads in addition to the main thread. One of these additional threads, called the *migration thread*, is used to pack the data together with the main thread arguments into a new closure to be migrated by the scheduler. The other thread is used to unpack the data from the closure upon reaching the destination.

In the following we shall explain how to use the protocol by writing an abstract search program. For simplicity, assume that the program searches a tree of fixed branching factor N, and that the state contained in a tree node is stored in a data structure of type State. The goal is to evaluate all the leaves of the tree (assuming results to be non-negative integers), and propagate the maximum leaf value to the root.

```
thread search (cont int k, State *p)
{
    if (isleaf (p)) accumulate (k, max, evaluate (p));
    else
    {   cont int k1;
        int i;

        spawn_next sum (k, ?k1{N: 0});
        for (i = 0; i < N, i++)
        {   State *q = generate_child(p, i);
            spawn search (k1, q, i);
        }
    }
    free_fstruct_local(p);
}
```

Note that one of the arguments to `search`, `p`, is a pointer. Normally this would be bad news for a thread, since `p` is valid only on the processor where the `search` thread was spawned. When the `search` thread is migrated to a different processor, `p` points to garbage — unless the data is also migrated with the thread, which is where the migration thread comes in.

A migration thread is specified by the attribute `migration` and has the same name and prototype as the main thread:

```
migration thread search (cont int k, State *p)
{
    closure *u;

    make_next_closure search_unpack (k, *p) {
        u = $;
    }
    free_fstruct_local(p);
    return u;
}
```

The mechanism of data migration is shown in Figure 2.6. When the closure of a `search`

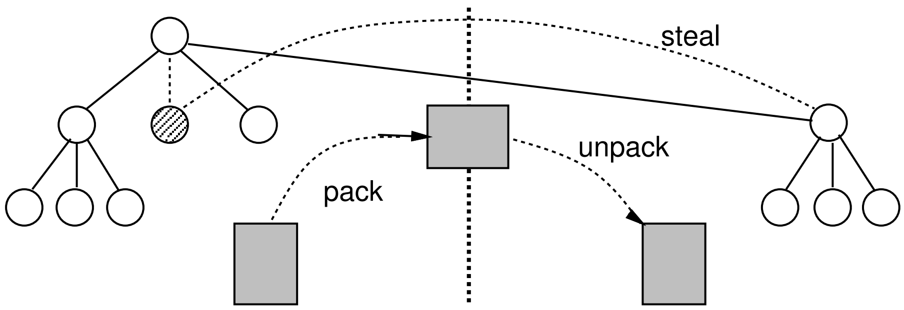

Figure 2.6: Packing and unpacking data during thread migration.

thread is chosen by work-stealing for migration, the runtime system calls the corresponding migration thread, which must pack the data into a new closure and return it immediately to the scheduler. In the migration thread for `search`, the statement

```
make_next_closure search_unpack (k, *p) {
    u = $;
}
```

is used to create the new closure. The Cilk keywords `make_closure` and `make_next_closure` are like `spawn` and `spawn_next`, except that the closure created is not posted to the scheduler. Note how the State structure pointed at by `p` is passed by value, so it is copied into the new closure. Within the trailing statement to `make_next_closure`, the special symbol `$` represents a pointer to the newly created closure, which can be captured in a local variable for the migration thread to return to the scheduler.

Note that the new closure is made for a different thread, search_unpack. When migrated and scheduled on the destination processor, the unpacking thread must reverse the actions of the migration thread, unpacking the data from the migration closure and calling search with a reference to the newly created data. The code for search_unpack is shown below:

```
thread search_unpack (cont int k, State s)
{
    State *p = alloc_fstruct_local (sizeof(State));
    *p = s;
    call_next search (k, p);
}
```

2.5 More Advanced Techniques

In the previous sections we have described how to write Cilk programs without having to worry about scheduling. In this section we shall describe some Cilk features that allow the programmer to control some or all of the scheduling decisions.

2.5.1 Overriding Work-Stealing

In combination with the Cilk work-stealing scheduler, it is sometimes useful to force certain threads or procedures to run on specific processors. For instance, if a data structure is statically laid out on your machine, you may want to post a thread to the processor containing the data it needs.

The statement

```
spawn foo (...) @pn;
```

where foo can be either a Cilk procedure or thread, posts foo on the processor numbered pn. The following code is an example of how to use this construct:

```
thread spawner (cont k, int P)
{
    int i;
    for(i = 0; i < P; i++)
    spawn worker (k) @i;
}
```

An explicitly posted thread or procedure becomes local (unstealable) on the destination processor. Thus, the pre-defined variable Self (which is always set to the local processor's number)can be used to force a closure to be executed locally.

```
spawn foo (...) @Self;
```

You can also specify in a thread's declaration that it should never be subject to work-stealing. A local thread is specified by the attribute local. For example, the thread defined as

```
local thread sum (cont k, int x, int y) { send_argument (k, x+y); }
```

is not subject to migration by work-stealing, i.e., it always executes on the processor where it is spawned.

2.5.2 Handler Threads

A handler thread is specified by the attribute `handler`. Handler threads exist as a low level mechanism to bypass the runtime scheduler. The runtime system keeps a separate queue for these threads so that they can be executed with high priority, i.e., before any threads in the main ready queue are executed.

In the following example, we illustrate how to use a handler thread to implement what are known as "split-phase" memory fetches. In this situation, the thread that initiates the fetch and the thread that uses the fetched data are separate. Assuming `p` to be a global pointer to an integer, it is shown as the following segment of code:

```
spawn_next use_data (..., ?k1, ...);
spawn fetch (k1, offset(p)) @(pn(p));
```

We define `fetch` to be a handler thread to minimize the latency involved:

```
handler thread fetch (cont int k, int *lp) { send_argument (k, *lp); }
```

Note that a handler thread is not subject to normal scheduling via work-stealing. Unless posted to another processor as in the previous example, it remains a local thread.

2.5.3 Entering the Scheduler from a SPMD Computation

In actual implementations, a Cilk program is first preprocessed to ordinary C code, which executes on all processors in SPMD mode. Multithreaded execution of Cilk procedures and threads only happens within each run of the scheduler. The simplest way to enter the scheduler is to do so automatically via the `cilk_main` procedure, but this technique requires the entire program to be written in mutithreaded style.

To be able to combine the benefit of the SPMD (data-parallel) style of computation with the multithreaded computation supported by the Cilk scheduler, one can explicitly enter the Cilk scheduler from within a data-parallel C program. In this case, the `cilk_main` procedure must not be defined. Instead, the following runtime system interface must be used:

```
int cilk_active_size;
int cilk_mode;

void CilkInit (int *pargc, char **argv);
void CilkExit (int exit-code);

[result =] schedule proc (arg_1, ...);
```

The following shows an example of typical use:

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

    CilkInit (&argc, argv);

    n = atoi (argv[1]);
    result = schedule fib (n);
    printf ("result: %d\n", result);

    CilkExit (0);
}
```

Before entering the scheduler, its state must be initialized by calling

```
CilkInit (&argc, argv);
```

where `argc` and `argv` are arguments supplied to the C `main` function. Since `CilkInit` processes the standard Cilk runtime system options, this must happen before any of the user options are processed. The function `CilkExit` should be used to clean up the scheduler state and terminate the program.

After initialization, the Cilk scheduler is entered via the construct

```
result = schedule fib (n);
```

The `schedule` statement is in fact part of the Cilk language, whose effect is to enter the scheduler and run `fib (n)` as the first procedure. `schedule` can only be used for this purpose. However, it may be used multiple times within the main control of the SPMD C program.

The global variable `cilk_active_size` can be set and reset before entering the scheduler to change the number of processors to participate in the computation. `cilk_mode` can be set before entering the scheduler to control the scheduler modes and is something not available when using `cilk_main`. The following modes are supported:

`CILK_AUTO`, `CILK_AUTO_KEEP`, `CILK_MANUAL`, `CILK_MANUAL_KEEP`,

In the `CILK_AUTO` mode, the Cilk runtime system uses work stealing, which is also the default mode. In the `CILK_MANUAL` mode, work stealing is turned off. In either mode the data heap is reset when the scheduler exits. If one wishes to keep the data structures allocated in the heap, which may be accessible either through the result pointer or some global variable set during computation, then `CILK_AUTO_KEEP` or `CILK_MANUAL_KEEP` should be used.

2.5.4 Data Structure Primitives

The Cilk runtime system provides a simple storage allocator for local data objects which can be used to create per-processor global data structures together with migration handlers, or distributed global data structures by creating global references to these objects. The primitives

```
void *alloc_fstruct_local (int size);
void free_fstruct_local (void *block);
```

allocate and free blocks of memory, where *size* is in units of words. The storage allocated is taken from the Cilk runtime heap, which is reset after each run of the scheduler by default. Therefore if one wishes to keep them over to the next run of the scheduler, before the current run the `cilk_mode` variable must be set to either `CILK_AUTO_KEEP` or `CILK_MANUAL_KEEP`.

2.5.5 Manual Scheduling in Cilk

In conjunction with the `CILK_MANUAL` mode, which turns off work-stealing, the `@dest` annotation can be used to schedule a computation by explicitly posting all threads. To illustrate, we shall consider a simple "shared" memory program using the Cilk storage allocator.

The shared memory abstraction we consider here is very primitive. It supports locally allocated data objects and global pointers. A global pointer (type `globptr`) is made of two components: a processor number (`pn`) and a local pointer (`offset`), which can be manipulated via the following interface:

```
globptr make_globptr (void *lp);
int pn (globptr p);
void *offset (globptr p);
```

(Note that `globptr` is not part of the Cilk language. One way to implement it uses a structure containing a processor number and a local pointer.)

Our example program makes a full binary tree and then traverses the tree to count the number of leaves. The data structure representing a node of the tree is defined as follows:

```
typedef struct {
    globptr left;
    globptr right;
} node;
```

The following code shows the Cilk procedure that makes the binary tree. It calls itself recursively to make two subtrees of height `h` − 1, then allocates a node above them to form a tree of height `h`. Note that we do not specify where each invocation of `make_tree` should be executed. We rely on work-stealing to distribute the allocation of the tree nodes across the processors.

```
cilk globptr make_tree (int h)
{
    if (h==0) return make_globptr(0); /* null pointer indicates leaf */
    else
    {
        globptr left, right;
```

```
        node *np;

        left = spawn make_tree(h-1);
        right = spawn make_tree(h-1);

        sync;
        np = (node *) alloc_fstruct_local (2);
        np->left = left;
        np->right = right;
        return make_globptr (np);
    }
}
```

Now for the count_leaves procedure. Unlike make_tree, we must explicitly post the count_leaves procedures to follow the global pointers:

```
cilk int count_leaves (globptr p)
{
    node *tree = (node *) offset (p);
    if (!tree) return 1;
    else
    {   globptr left = tree->left, right = tree->right;
        int x, y;

        x = spawn count_leaves (left) @(pn(left));
        y = spawn count_leaves (right) @(pn(right));

        sync;
        return x+y;
    }
}
```

The following code is the top level control of the program. The scheduler is entered twice: the first time to run make_tree, the second time to run count_leaves. We enter the scheduler twice so that we can switch the scheduler mode to MANUAL between the first and second calls, which turns off work-stealing for the second call:

```
int main (int argc, char *argv[])
{
    int h, result;
    globptr tree;

    CilkInit (&argc, argv);
    h = atoi (argv[1]);
```

```
    cilk_mode = AUTO_KEEP;
    tree = schedule make_tree (h);

    cilk_mode = MANUAL;
    result = schedule count_leaves (tree);

    CilkExit (0);
}
```

The mode `AUTO_KEEP` indicates to the scheduler to keep the data structures allocated via `alloc_fstruct_local` around for the next run of the scheduler (see section 2.5.3).

Chapter 3

Shared Memory

New to Cilk with version Cilk 3.0 is dag-consistent shared memory. For a complete description of the shared memory consistency model, its performance, and its provable properties, see [BFJ+95][1]. Here, we give a brief overview of the consistency model and describe how to program using Cilk's shared memory.

3.1 Overview of Consistency Model

The best way to explain Cilk's dag-consistent shared memory is by example. To illustrate the concepts behind dag consistency, consider the problem of parallel matrix multiplication. One way to program matrix multiplication is to use the recursive divide-and-conquer algorithm shown in Figure 3.1. To multiply one $n \times n$ matrix by another, we divide each matrix into four $n/2 \times n/2$ submatrices, recursively compute some products of these submatrices, and then add the results together. This algorithm lends itself to a parallel implementation, because each of the eight recursive multiplications is independent and can be executed in parallel.

Figure 3.2 shows Cilk code for a *blocked* implementation of recursive matrix multiplication in which the (square) input matrices `A` and `B` and the output matrix `R` are stored as a collection of 16×16 submatrices, called *blocks*. All three matrices are stored in shared memory. The Cilk procedure `matrixmul` takes pointers to the first block in each matrix as input, as well as a variable `nb` denoting the number of blocks in any row or column of the matrices. From the pointer to the first block of a matrix and the value of `nb`, the location of any other block in the matrix can be quickly computed. As `matrixmul` executes, values are stored into `R`, as well as into a temporary matrix `tmp`.

The procedure `matrixmul` operates as follows. Lines 3–4 check to see if the matrices to be multiplied consist of a single block, in which case a call is made to a serial routine `multiply_block` (not shown) to perform the multiplication. Otherwise, line 8 allocates some page-aligned temporary storage in shared memory for the results, lines 9–10 compute pointers to the 8 submatrices of `A` and `B`, and lines 11–12 compute pointers to the 8 submatrices of `R` and the temporary matrix `tmp`. At this point, the divide step of the divide-and-conquer

[1] Available via ftp from `ftp://theory.lcs.mit.edu/pub/cilk/dag-consistent.ps.Z`

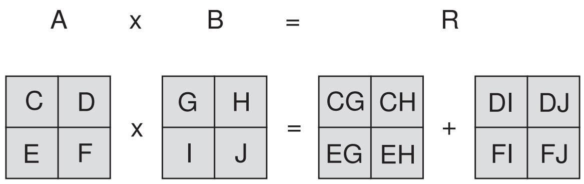

Figure 3.1: Recursive decomposition of matrix multiplication. The multiplication of $n \times n$ matrices requires eight multiplications of $n/2 \times n/2$ matrices, followed by one addition of $n \times n$ matrices.

paradigm is complete, and we begin on the conquer step. Lines 13-20 recursively compute the 8 required submatrix multiplications, storing the results in the 8 disjoint submatrices of `R` and `tmp`. After the `sync`, line 22 spawns a parallel addition in which the matrix `tmp` is added into `R`. (The procedure `matrixadd` is itself implemented in a recursive, parallel, divide-and-conquer fashion, and the code is not shown.) The `sync` in line 23 ensures that the addition completes before `matrixmul` returns.

Like any Cilk multithreaded computation [BJK+95], the parallel instruction stream of `matrixmul` can be viewed as a directed acyclic graph (dag) of threads organized into a tree of procedures. Figure 3.3 illustrates the structure of the dag. Each vertex corresponds to a thread of the computation. The syncs in lines 21 and 23 break the procedure `matrixmul` into three threads X, Y, and Z, which correspond respectively to the partitioning and spawning of subproblems $M_1, M_2, \ldots, M_8$ in lines 2–20, the spawning of the addition S in line 22, and the return in line 25. The dag of a Cilk computation contains three kinds of edges. A *spawn* edge connects a thread with its spawned child. A *continue* edge connects a thread with its successor in the same procedure. A *return* edge reflects the synchronization that occurs when a child completes and notifies the thread in its parent procedure that is waiting for its return. Thus, a Cilk computation unfolds as a *spawn tree* composed of procedures and the spawn edges that connect them to their children, but the execution is constrained to follow the precedence relation determined by the dag of threads.

What kind of memory consistency is necessary to support a shared-memory program such as `matrixmul`? Certainly, sequential consistency can guarantee the correctness of the program, but a closer look at the precedence relation given by the dag reveals that a much weaker consistency model suffices. Specifically, the 8 recursively spawned children $M_1, M_2, \ldots, M_8$ need not have the same view of shared memory, because the portion of shared memory that each writes is neither read nor written by the others. On the other hand, the parallel addition of `tmp` into `R` by the computation S requires S to have a view in which all of the writes to shared memory by $M_1, M_2, \ldots, M_8$ have completed.

The basic idea behind dag consistency is that each thread sees values that are consistent with some serial execution order of the dag, but two different threads may see different serial orders. Thus, the writes performed by a thread are seen by its successors, but threads that are incomparable in the dag may or may not see each other's writes. In `matrixmul`, the computation S sees the writes of $M_1, M_2, \ldots, M_8$, because all the threads of S are successors of $M_1, M_2, \ldots, M_8$, but since the M_i are incomparable, they cannot depend on seeing each others writes. For a formal model of dag consistency, see [BFJ+95].

```
cilk void matrixmul(long nb, shared block *A, shared block *B,
                             shared block *R)
 {
  if (nb == 1)
    multiply_block(A, B, R);
  else {
    shared block *C,*D,*E,*F,*G,*H,*I,*J;
    shared block *CG,*CH,*EG,*EH,*DI,*DJ,*FI,*FJ;
    shared page_aligned block tmp[nb*nb];

    /* get pointers to parts of original matricies */
    partition_matrix(nb, A, &C, &D, &E, &F);
    partition_matrix(nb, B, &G, &H, &I, &J);

    /* get pointers to places to put results */
    partition_matrix(nb, R, &CG, &CH, &EG, &EH);
    partition_matrix(nb, tmp, &DI, &DJ, &FI, &FJ);

    /* do multiplication subproblems */
    spawn matrixmul(nb/2, C, G, CG);
    spawn matrixmul(nb/2, C, H, CH);
    spawn matrixmul(nb/2, E, H, EH);
    spawn matrixmul(nb/2, E, G, EG);

    spawn matrixmul(nb/2, D, I, DI);
    spawn matrixmul(nb/2, D, J, DJ);
    spawn matrixmul(nb/2, F, J, FJ);
    spawn matrixmul(nb/2, F, I, FI);
    sync;

    /* add results together into R */
    spawn matrixadd(nb, tmp, R);
    sync;
  }
return;
}
```

Figure 3.2: Cilk code for recursive blocked matrix multiplication. The call `multiply_block(A,B,R)` performs a serial multiplication of blocks `A` and `B` and places the result in block `R`. The side length of each of these three matrices is `nb` blocks. The procedure `matrixadd` is implemented by a straightforward parallel divide-and-conquer algorithm.

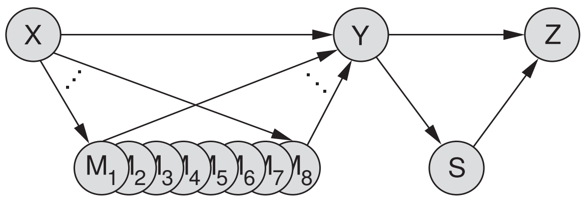

Figure 3.3: Dag of blocked matrix multiplication. Each circle represents a thread of the computation. Threads are linked by downward spawn edges, horizontal continue edges, and upward return edges. Some edges were omitted for clarity.

3.2 Programming in Shared Memory

Cilk provides some linguistic abstractions to make programming in shared memory easy. A brief outline of these constructs is given below. For a full description of the constructs, see Section 4.12.

3.2.1 Shared Pointers

Objects in shared memory can be referenced using `shared` pointers. For example, in Figure 3.2, each of the three matrices to be multiplied is referenced using a `shared` pointer. Any operation legal on a C pointer is also legal on a shared pointer, including indirection (`*`), array reference (`[ ]`), structure reference (`->`), increment/decrement operators, assignment operators, additive operators, relational operators, and equality operators. Shared pointers point to a different address space than C pointers, however, so casting a shared pointer to a C pointer or vice versa is illegal.

As an example of using `shared` pointers, consider implementing the function `multiply_block` in Line 4 of the `matrixmul` code. One implementation would be as follows:

```
typedef struct {
     double elems[BLOCK_SIZE][BLOCK_SIZE];
} block;

void multiply_block (shared block *A, shared block *B,
                     shared block *R)
{
    int i,j,k;
    for(i=0; i<BLOCK_SIZE; i++)
         for(j=0; j<BLOCK_SIZE; j++) {
              R->elems[i][j] = 0.0;
              for(k=0; k<BLOCK_SIZE; k++)
                   R->elems[i][j] += A->elems[i][k] *
                                     B->elems[k][j];
         }
}
```

Shared pointers are fast for doing pointer arithmetic and index calculation, but slow for dereferencing. A `shared` pointer dereference costs on the order of 100 cycles because a page fault check must be done in software on each dereference (see Section 3.4 for more detailed performance numbers).

3.2.2 Register Shared Pointers

Because `shared` pointers are slow to dereference, we also provide `register shared` pointers. These pointers are fast when multiple accesses occur to the same page. A few restrictions are put on the use of `register shared` pointers. The first two are the same restrictions that C puts on `register` objects, and the others are additional for `register shared` pointers.

- Only formal parameters and local (automatic) variables can be `register shared` pointers.
- It is illegal to take the address of a `register shared` pointer.
- A `register shared` pointer can only access memory less than `CSM_REGISTER_LIMIT` bytes away from its last assignment (its last setting by formal binding or the `=` operator). Currently, `CSM_REGISTER_LIMIT` is set to almost 1 gigabyte.
- A `register shared` pointer potentially locks down a page in the page cache whenever it is in scope, so having many `register shared` pointers in scope can hurt performance.
- `register shared` pointers use more machine registers than regular `shared` pointers, and thus may create additional register scheduling demands on the compiler.

In contrast to regular `shared` pointers, `register shared` pointers cost only a few (6 or 7) cycles to dereference when the previous dereference of that pointer was also to the same page. `register shared` pointers are useful when accessing an array or object in shared memory with high spatial locality. For instance, in the `multiply_block` example above, if we replace the declaration of `multiply_block` with

```
void multiply_block (register shared block *A, register shared block *B,
                     register shared block *R)
```

then all accesses to `A`, `B`, and `R` will be done using `register shared` pointers.

3.2.3 Converting Shared Pointers to C Pointers

For performance-critical sections of code, we also provide a mechanism to convert shared pointers to C pointers. Although this mechanism allows for accesses to proceed at the full speed of the underlying architecture, it must be used with caution. We provide the following three calls:

```
const void *pin_page(const shared void *p);
void *pin_page_write(shared void *p);
void unpin_page(const shared void *p);
```

The functions pin_page and pin_page_write convert a shared pointer to a C pointer. A pointer returned by pin_page may only be read from, whereas a pointer returned by pin_page_write can be both read and written. In either case, the pointer returned can only be used to access the page of data to which the shared pointer was pointing. Any access outside this page is considered an error. The page size is given by the constant CSM_PAGE_SIZE, so the pointer returned is only valid to access the addresses
`(shared void*)((long)p & ~(CSM_PAGE_SIZE-1))`
through
`(shared void*)(((long)p & ~(CSM_PAGE_SIZE-1)) + CSM_PAGE_SIZE - 1)` .
Finally, each call to pin_page or pin_page_write must be matched by a unpin_page call with the same shared pointer as its argument.

We can implement multiply_block using these low-level calls as follows:

```
void multiply_block (shared block *A, shared block *B,
                     shared block *R)
{
    int i,j,k;
    block *a = (block*)pin_page(A);
    block *b = (block*)pin_page(B);
    block *r = (block*)pin_page_write(R);

    for(i=0; i<BLOCK_SIZE; i++)
         for(j=0; j<BLOCK_SIZE; j++) {
              r->elems[i][j] = 0.0;
              for(k=0; k<BLOCK_SIZE; k++)
                   r->elems[i][j] += a->elems[i][k] *
                                     b->elems[k][j];
         }
    unpin_page(A);
    unpin_page(B);
    unpin_page(R);
}
```

Note that we must ensure that no block crosses a page boundary for this code to be correct. We can enforce this condition by choosing BLOCK_SIZE to make the size of a block no larger than a page, and use the page_aligned directive (see section 4.12.1) to make sure a block does not cross a page boundary.

3.3 Allocation

Shared memory objects can be allocated using the shared keyword in front of declarations, such as that shown in Line 8 of Figure 3.2, or by using the allocate_stmem call. Both of these allocation mechanisms are stack-like allocation, corresponding to allocation on the stack by the compiler and explicitly calling the C library routine alloca, respectively. We currently have no heap allocator for Cilk shared memory.

The simplest method for allocating a shared memory array is to simply declare a particular array as shared. For example, in Figure 3.2, an array of blocks called tmp is allocated in Line 8. The identifier tmp can then be used as a shared pointer to the first element of that array. Unlike C array declarations, shared array declarations can use bounds determined at run-time, providing a versatile allocation mechanism. Arrays allocated in this fashion are automatically deallocated when leaving their scope.

Shared memory can also be allocated using the allocate_stmem call. The declaration of allocate_stmem is as follows:

```
shared void *allocate_stmem(SharedSizeT size, int alignment);
void free_stmem(shared void *p);
```

allocate_stmem allocates size bytes of storage aligned to alignment bytes[2]. free_stmem frees a piece of storage allocated by allocate_stmem. Note that allocation is done on a stack, so freeing a memory location also frees all memory allocated after that location was allocated.

Because allocation is stack-based, a pointer to any memory allocated by a child cannot be passed to its parent. Deallocation is automatically performed upon the return of the child, as is stack memory in C. Any information passed from the child to its parent must be done via shared memory that the parent (or one of its ancestors) allocated.

3.4 Performance

This section compares the performance of three mechanisms for accessing shared memory: (1) explicit page pinning and ordinary C pointers; (2) shared pointers; and (3) register shared pointers.

Figure 3.4 shows timings for shared memory accesses on the CM5. The timings assume that the page is found in the local processor's page cache, and the word being accessed is in the memory cache. (Page misses generally cost on the order of 3 milliseconds, or 100,000 cycles.)

As the table indicates, accessing shared memory through a shared pointer incurs a significant penalty: 76 cycles for a shared-memory load under best conditions, compared to only 2 cycles for a local-memory load. Accessing a word of shared memory through a register shared pointer, however, costs only 5 cycles more than an ordinary memory access.

Register shared pointers suffer a greater penalty for visiting a new page, because of the cost of pinning and unpinning. If this cost can be amortized over several accesses to the page, then a register pointer is a better choice. The table suggests that just two accesses to the same page are sufficient to make a register pointer faster than a shared pointer. Two reads through a shared pointer cost $76 \times 2 = 152$ cycles, whereas two reads of the same page through a register pointer cost only $124 + 7 = 131$ cycles.

[2]Use the constant CSM_PAGE_SIZE as the alignment to ensure that the allocated memory is page-aligned.

Pointer	Operation	Cycles
C pointer (to pinned page)	Load	2
	Store	3
Shared pointer (page found in cache)	Load	76
	Store	84
Register shared pointer (page already pinned)	Load	7
	Store	8
Register shared pointer (page not pinned but found in cache)	Load	124
	Store	131

Figure 3.4: Timing of memory access operations on the CM5. Times are for single-word loads or stores, assuming no misses in the processor's memory cache.

Register pointers have other costs which must be considered, however. Assignment is more expensive for a register shared pointer, since arithmetic is required. Assigning to a register pointer costs 32 cycles on the CM5 (compared to 1 for shared pointers and C pointers). Also, the presence of a register pointer increases register pressure, since it uses two more local variables than the other pointers.

3.5 Command-Line Options

When Cilk is compiled with shared memory enabled, two additional command-line options are available:

`-cache` *size* Set the size of the page cache on each processor to *size* megabytes. The default is 1 megabyte. *size* may be specified in floating-point format. We recommend that approximately one quarter of the available primary memory of each processor be allocated to the page cache.[3]

`-bstore-proc` *n* Use *n* processors to hold backing store. The default is to use as many processors as specified by the `-nproc` option. By setting the number of backing store processors larger than the number of working processors, a job requiring a large amount of memory can be run on a small number of processors.

These options can be specified anywhere on the command line that a regular Cilk option can be specified (see Section 2.2.4).

3.6 Compatibility of Shared Memory with other Cilk features

Unfortunately, our shared memory implementation depends strongly on the fully strict behavior of programs written with Cilk procedures. As a consequence, certain low-level schedul-

[3]Half of the primary memory should be reserved for backing store, and the last quarter should be used for OS, code, RTS data structures, etc.

ing constructs conflict with the operation of the shared memory and can cause the system to crash. These low-level scheduling constructs include:

`local` and `handler` threads Threads declared `local` or `handler` cannot be stolen and thus destroy the provable properties of the Cilk scheduler. These types of threads can be used *provided* that they do not make any shared-memory references and do not spawn or enable any other regular threads or procedures.

the `@` directive The `@` directive creates a local thread or procedure, so it has the same prohibition as a local thread above.

Explicit Continuation Passing The shared memory system requires all `send_argument`s to go to the parent procedure. One can violate this requirement by using continuation passing explicitly. If you use explict continuation passing, make sure that all your `send_argument`s go only to the parent procedure.

Scheduling Mode The scheduling mode `cilk_mode` cannot be set to `CILK_MANUAL` or `CILK_MANUAL_KEEP`.

We are currently working on pinpointing the exact subset of Cilk that may be used with shared memory and detecting at run-time when constructs outside this subset are used. For now, however, Bad Things^{TM} happen when incompatible constructs are used. Be advised.

Chapter 4

Language Reference

4.1 Basic Concepts

In Cilk, the basic unit of scheduling and execution is a *thread*. Threads are grouped together to form *procedures*, which are the parallel equivalent of C functions. Procedures can be directly defined in Cilk, whithin such a procedure threads are punctuated by synchronization points. Threads can also be directly defined in Cilk, in which case the notion of a procedure is maintained through a system of annotations in the Cilk language.

To avoid confusion of terminology, we shall use the terms "Cilk thread" and "Cilk procedure" to refer to the directly defined threads and procedures, and use the terms "thread" and "procedure" in a more abstract sense. At runtime, procedure and thread spawning forms a dynamic tree, with data dependencies among threads forming a dag (see section 1.3).

While Cilk procedures form an abstraction that most C programmers will feel comfortable with, Cilk threads provide an abstraction that is more primitive but necessary for direct access of some runtime mechanisms.

4.1.1 Closures and Continuations

A Cilk thread is spawned by making a closure. A closure is a data structure that captures all information needed to execute the thread, containing the thread pointer and all arguments passed to the thread. To enable thread communication and synchronization, a closure can be created waiting for some arguments to arrive in the future, thus it may contain empty argument slots. A closure is said to be *full* if it has all of its arguments, otherwise it is *waiting*.

A *continuation* is a global reference to an empty argument slot, to which an argument can be sent. For example, the statement

```
spawn foo (k, ?x, ?y);
```

creates a closure for `foo`, and initializes `x` and `y` as continuations pointing to the two empty slots:

[1] This chapter was written by Yuli Zhou and Rob Miller.

	foo
	k
x →	
y →	

Usually, both x and y will be passed to new threads which will eventually send arguments via

```
send_argument (x, arg);
```

that fills the slot referenced by x. The closure will be *posted* to the scheduler when it becomes full, which will later schedule it for execution.

4.1.2 Frames

When a Cilk procedure is spawned, an activation *frame* for the procedure is made and given to the scheduler. A frame is a closure shared by the initial and successor threads in the same procedure, thus from a runtime system point of view, frames are just extended closures. In this document, we shall often use the term "closure" to also refer to frames for procedures. The Cilk preprocessor breaks down Cilk procedures into threads. At runtime, the thread pointer in the frame is initialized to the first thread in the procedure, then mutated to the successor threads as computation unfolds.

The continuation-passing linkage of procedures is handled by the Cilk preprocessor and hidden from the programmer. Specifically, when spawning a procedure, the return continuation (refering to the return slot in the parent frame) is passed as an implicit first argument. Within the procedure, the statement

```
return val;
```

is used to send the return value using the implicit return continuation, which also has the effect of terminating the current procedure execution.

4.1.3 Runtime Scheduling and Work Stealing

As we mentioned in section 1.3, 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 ready queue per processor that keeps closures by their spawning depth. Closures for successor threads are posted at the same level, while those for child threads are always posted to the next level. For local execution, the scheduler always takes the most recently spawned closure 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 ready queue becomes empty, the processor attempts to steal a closure from a randomly chosen "victim." The closure chosen for the steal is the most recently spawned closure in the shallowest level of the victim's ready queue. Intuitively, a thread in the shallowest level is likely to contain more work, so the thief processor will have more work to do before it must steal again.

4.2 Program Structure

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

Within a Cilk program, C code and C functions are used for sequential computation. C functions can be called from Cilk procedures and threads, but Cilk procedures and threads cannot be invoked from C functions. The only exception to this rule is a sequential `call` to a Cilk procedure, which treats the procedure exactly like a C function (i.e., it does not run in parallel, and all of its computation is sequential).

4.3 Keywords

The following words have special meaning in Cilk, and thus cannot be used as ordinary identifiers in Cilk code:

```
accumulate, byvalue, call, call_next, cilk, cilk_main, closure, cont,
handler, local, make_closure, make_next_closure, migration,
schedule, send_argument, signal, spawn, spawn_next, sync,
tail_call, tail_call_next, thread, shared, page_aligned
```

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 with extension `.c` or `.h` (the conventional extensions for C source). The Cilk preprocessor treats files with these extensions as straight C. Thus, standard 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 should be used when the C code is found within a Cilk program (`.cilk`) file.

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

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

4.4 Types

The thread abstraction in Cilk introduces the following new types: `cont`, `closure` and `signal`. With the exception of `signal`, which has special restrictions, these types may appear anywhere in a Cilk program, including ordinary C functions and declarations.

`cont` is a type qualifier used to declare continuations. Continuation declarations have the general form

`cont` *type var*

where `type` is called its *base type*. A continuations is a reference to a closure slot having the based type, thus its usage is similar to the C "`*`" (pointer-to) type qualifier. Continuations are captured at thread spawning time (section 4.7.2), and subsequently passed to other threads to enable values to be sent to the referenced slot (section 4.7.3).

`signal` is a special type, which is introduced for the frequent cases where a thread only needs to receive a notification from another thread which carries no value. Technically, `signal` specifies a 0-sized slot in the closure, thus continuations can be defined to refer to a signal slot (section 4.7.2). Since the signal carries no value, only the continuation is significant: the signal itself can never be used. A signal only serves as a place holder in the thread argument declaration list for the purpose of type-checking.

4.5 Procedure/Thread Definitions and Declarations

Cilk extends ANSI C with constructs to define procedures and threads:

procedure: `cilk` *type proc* `(`*arg-decl*$_1$, ..., *arg-decl*$_n$`)` *body*
thread: [*attrib*] `thread` *thr* `(cont` *type k, arg-decl*$_1$, ..., *arg-decl*$_n$`)` *body*

The keyword `cilk` specifies a procedure definition, and the keyword `thread` specifies a thread definition. Argument declarations follow the ANSI C convention. When the body of the definition is the null statement (`;`), both forms are taken to be prototype declarations.

Note that procedures have a return type, but threads have an explicit continuation argument (appearing first) whose base type is the return type of the thread. An empty or `void` argument list specifies a procedure or thread with no argument.

The optional *attrib* field to a thread definition can be one of

`local`, `handler`, `migration`

The use of these keywords will be described later in section 4.10 and 4.11.

Procedure and thread bodies take the form of a C compound statement:

$\{\ decl_1 \ldots stmt_1 \ldots \}$

Here the declarations follow the ANSI C convention. All ANSI C statement and expression syntax are allowed in the procedure or thread body, with the following exception: *New types may not be defined in a procedure body.* Thus, all `struct`, `union`, and `enum` definitions and `typedef` declarations used in a Cilk program should appear outside of procedure bodies.

Procedure and thread bodies may also contain Cilk statements, including procedure and thread spawns, synchronization, and communication.

4.6 Spawning

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

stmt: [*var* =] `spawn` *proc_or_thread* `(` arg_1, ..., arg_n`)` [`@(`*expr*`)`] *stmt*
`spawn`[`_next`] *proc_or_thread* `(` *k*, arg_1, ..., arg_n`)` [`@(`*expr*`)`] *stmt*

The first form, the implicit continuation-passing form, is used to spawn procedures or threads from a Cilk procedure. The left-hand side *var* `=` must be present if and only if *proc_or_thread* has a non-void return type. *var* must be a local variable, or an element of a locally declared union, structure or array. When *proc_or_thread* is a procedure, *var* must have the same type as the return type of the procedure. When *proc_or_thread* is a thread, *var* must have the same type as the base type of thread's initial continuation argument.

The second form, the explicit continuation-passing form, is used to spawn procedures or threads from a Cilk thread. The keyword `spawn` should be used to create a child thread that starts a new procedure invocation, whereas `spawn_next` should be used to create a successor thread within the same logical procedure. When *proc_or_thread* is a procedure, the continuation argument *k* must have the same base type as the return type of the procedure. When *proc_or_thread* is a thread, *k* must have the same continuation type as the thread's initial continuation argument.

In either form, *proc_or_thread* must be defined or declared as a Cilk procedure or thread with the correct number and types of arguments for arg_1, ..., arg_n.

Spawning a Cilk procedure or thread creates a closure. If the optional `@(`*expr*`)` is present, the closure will be posted to the processor number *expr*, otherwise it will be posted locally. If all arguments are available then posting happens immediately.

The trailing *stmt* can be null (`;`) or any Cilk statement. Since it is executed before the closure is posted, the trailing statement is handy for initializing the closure. See section 4.7.1 for more details.

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

4.7 Passing Arguments and Returning Values

Arguments passed to spawned procedures or threads can be any valid C expression. In addition, Cilk provides extensions to pass arrays by value (section 4.7.1) and to capture continuations referring to slots in waiting closures (section 4.7.2).

Within a Cilk procedure, the 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.

Within a Cilk thread, the `return` statement merely terminates the thread's execution, without sending a value. Threads must communicate via explicit continuation-passing, using an assortment of primitives provided in the Cilk language (section 4.7.3).

4.7.1 Passing Array Arguments by Value

Cilk introduces the storage class `byvalue`, specified before an array argument declaration, to indicates that the array should be copied when it is passed. For example:

```
procedure queens (byvalue int path[], int i, int n) ...
```

Two kinds of pass-by-value array arguments can be declared:

arg-decl:	`byvalue` *var*`[`*expr*`]`	fixed-size array
	`byvalue` *var*`[]`	variable-size array

The actual size of a variable-size array argument is specified at thread spawning time. Variable-size array arguments can occur anywhere inside the argument list.

When spawning a procedure or a thread, there are two ways to supply actual arguments declared to have `byvalue` array type: with a *range expression* or with an *initialization pointer*.

A range expression passes a specified range of an existing array. It takes one of the following forms:

arg: *expr*`[..]`
expr`[`$expr_l$`..`$expr_u$`]`

The first form can only be passed to an argument declared to have `byvalue` array type of fixed-size, where the size of the array is taken from the declaration. The second form can only be passed to an argument having `byvalue` variable-size array type. The array segment passed includes elements at both the lower and upper bounds (i.e., the size of the array object is calculated as $expr_u - expr_l + 1$).

An initialization pointer takes one of the following forms

arg: `:`*var*`[]`
`:`*var*`[`*expr*`]`

The arguments corresponding to these forms must have `byvalue` array types of fixed and variable size, respectively. The size of the array is taken from the declaration in the first form, and is specified by *expr* in the second form. Either form has the effect of allocating an array of the specified size in the closure, and sets *var* to be a pointer to the beginning of the array.

The array allocated by an initialization pointer must be initialized explicitly in the spawn's trailing statement (see section 4.6). *var* is set to point to the first element of the array in the closure, and must be declared to match the array element type. Example:

```
int *new_path;
```

```
ncount = spawn queens (:new_path[i+1], i+1, n) {
              memcpy ((char*)new_path, (char*)path, i*sizeof(int));
              new_path[i] = j;
         }
```

Since the closure is automatically posted to the scheduler after executing the trailing statement, the initialization pointer should not be referenced after the spawn statement.

4.7.2 Continuation Capturing Arguments

For Cilk threads that are spawned with waiting closures, the following forms can be used to capture continuations referring to the empty argument slots:

arg : ?*var*
?*var*{$expr_j$: $expr_i$ }
?*var*{$expr_j$ }
?*var*[] | ?*var*[*expr*]

An argument of one of the following forms

?*var*
?*var*{$expr_j$: $expr_i$}
?*var*{$expr_j$ }

allocates an empty slot in the closure, and initializes *var* to a continuation referring to that slot. In all of the forms *var* must be declared to have the type `cont` *type*, where *type* is the type of the argument slot to which it points.

The argument forms here differ in how many argument values are expected and how the values are deposited in the empty slot. Specifically:

- ?*var* allocates a simple slot that waits for exactly one argument to be sent.
- ?*var*{$expr_j$: $expr_i$} allocates an *accumulator* slot, expecting $expr_j$ values and initializes the slot to $expr_i$. The closure is not considered full until all $expr_j$ values arrive. The means of accumulation of these values is unspecified; it depends on which primitives are used to send the arguments (see section 4.7.3).
- ?*var*{$expr_j$ } specifies a *signal* slot. The argument must be declared to have the type `signal`, and the slot has size 0 (in effect, no slot will be allocated in the closure for a signal). As in the previous case, the closure will be waiting for an additional $expr_j$ signals.

Both of the following argument forms

?*var*[]
?*var*[*expr*]

allocate an array of simple slots and initialize *var* to a continuation pointing to the beginning of the array. The first argument form must have `byvalue` fixed-size array type, and the size is taken from the type declaration. The second form must have `byvalue` variable-size array type. Continuation arithmetic can be used on *var* to derive continuations that reference other slots in the array (see section 4.7.4).

4.7.3 Thread Communication Primitives

Cilk threads communicate via explicit continuations using the following statements:

```
send_argument (k, v);
accumulate (k, op, v);
send_argument (k);
```

Here v is first coerced to the base type of the continuation k as if for assignment.

The first form is used to send an argument to a simple slot in a waiting closure. The third form is used to send a signal. The second form takes an additional *op* argument, which is used to send an argument to an accumulator slot, where *op* is an operator that specifies the accumulating operation.

There are six predefined accumulation operators:

```
add, sub, and, or, min, max
```

In addition to these predefined operators, a user-written C function may act as an accumulation operator. Such a function *op* must have type `void (`T `*`, T `*)`, where T is the base type of the continuation. The first argument is a pointer to the value being accumulated; the second argument is a pointer to the accumulator slot. For example, a user-defined integer add operator might look like:

```
void int_add (int *v, int *a) { *a += *v; }
```

User defined operators should be commutative and associative, since the order in which accumulations will occur is undefined.

4.7.4 Continuation Arithmetic

With thread creation, for example

```
cont int x;
spawn_next foo (1, ?x[SIZE]);
```

it is only possible to capture a single continuation `x` pointing to the first element of the array. In order to send the other elements of an array argument, it is necessary to index the continuation as you would index a C pointer:

```
send_argument(k + i, v );
```

In general, k `+` i returns a new continuation pointing to the ith element counting from the reference point of k, where the size of each element is the size of k's base type.

It is illegal to index a continuation that points to a closure on a remote processor.[2] In practice, a thread should only index continuations that it created, not continuations that were passed in by its parent. For example, the continuation arithmetic below is safe:

```
thread parent(cont int k, ...)
{
    cont int x;

    spawn_next parent2(k, ?x[3], ...);
    spawn child(x, ...);
    spawn child(x+1, ...);
    spawn child(x+2, ...);
}
```

but `child` may not change the continuation it receives from `parent`.

Continuation arithmetic may be performed with all the usual C operators for pointer arithmetic: `+`, `-`, `++`, `--`, `+=`, `-=`. Unlike pointers, however, continuations may not be subtracted to find their relative displacement.

4.8 Synchronization

Within the body of a Cilk procedure, the 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.

With dag-consistent shared memory enabled, the `sync` statement has additional meaning. The sync statement ensures that all writes to shared memory performed by the `spawns` preceeding 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.

Cilk threads are non-blocking, and thus cannot contain sync statements. Synchronization happens implicitly as the side-effect of sending arguments to threads, in the sense that a thread waits for all arguments to arrive before it is ready for execution.

[2] This is a consequence of the current implementation of the Cilk runtime system, which creates a stub for the return argument slot when it steals a computation.

4.9 Sequential Execution

In Cilk, both procedures and threads can be directly called as C functions. However, calling procedures is quite different from calling threads, as further described below.

4.9.1 Sequential Execution of Procedures

The following expression calls a Cilk procedure sequentially:

`call` *proc* `(` arg_1, ... `)`

The type of the expression is the return type of the Cilk procedure.

If a Cilk procedure is invoked in this way, all computation specified by the procedure, including child procedures that it spawns, are executed sequentially on the local C stack, i.e., exactly like C functions. This mechanism is useful for increasing the granularity of threads, thus reducing the scheduling overhead, at the potential cost of reduced parallelism.

4.9.2 Direct and Tail Calling Threads

The following statement forms can be used to call threads:

`call`[`_next`] *thr* `(` *k*, arg_1, ... `);`
`tail_call`[`_next`] *thr* `(` *k*, arg_1, ... `);`

`call` invokes the thread *thr* exactly as a C function. `tail_call` is more efficient, but is restricted in that *thr* must be the very thread in whose body it appears[3]. Both the direct and tail calls avoid making a closure and completely bypass the scheduler. The `_next` suffix has the same effect of specifying a successor thread as does `spawn_next`. Unlike calling Cilk procedures, a direct or tail called thread may still spawn threads that can execute in parallel.

4.9.3 Notes on Argument Passing

When calling Cilk procedures or threads as C functions, arguments passed to them must be C expressions as no closures are created. Specifically, no continuation-capturing arguments can be specified when calling a thread. In addition, a `byvalue` array argument must be passed by reference. For the latter, the Cilk preprocessor generates C code that is consistent with this interpretation of array arguments, i.e., if we have a procedure declared as

```
cilk int foo (byvalue int a[10]) { ... }
```

and `b[10]` is an array, then `b[10]` will be copied into the frame when `foo` is spawned, but only the pointer `b` will be passed to `foo` when it is called.

If an argument is a structure, then it is passed by value no matter whether the procedure is spawned or called. When a structure is large, one may wish to avoid the extra copying during direct calls. This can be achieved by specifying it as an array argument of 1 element. For example, instead of defining

[3] A tail call is implemented as a jump back to the entry of the thread after resetting the arguments.

```
cilk int foo (struct bar x) { ... }
```

and having to pass its `struct` argument by value when `foo` is called, change the definition to

```
cilk int foo (byvalue struct bar xp[1]) { ... }
```

4.10 Annotations for Thread Scheduling

Under the default Cilk scheduling policy, the programmer specifies neither where nor exactly when a thread will be executed: it is up to the scheduler to execute the threads as they become ready or migrate them upon work stealing requests.

The different versions of `spawn` and `spawn_next` should be regarded as annotations providing hint to the scheduler so that it can group together threads logically belonging to the same procedure invocation for scheduling purposes (see section 4.1.3).

For more complicated situations, Cilk provides a more elaborate system of annotations by which the programmer can post threads to specific processors, to make some threads not stealable, or to execute certain threads at an higher priority.

The form

`local thread` *thr* `(`*arg-decl*$_1$`, ...)` *body*

defines a thread that stays local, i.e., it will not be migrated by work stealing. The statement

`spawn[_next]` *thr* `(`*arg*$_1$`, ...) @(`*expr*`)` *stmt*

creates a closure which will be posted to the processor numbered *expr*. Upon arrival it becomes a local closure on the destination processor. Consequently, the closure created by

```
spawn foo (...) @(Self)
```

will not be stolen (`Self` is a global variable defined in the runtime system, whose value is the local processor's number).

The form

`handler thread` *thr* `(`*arg-decl*$_1$`, ...)` *body*

defines a handler thread. A handler thread is similar in every respect to an ordinary thread except that it has higher scheduling priority, i.e., once ready it is scheduled to execute before all the ordinary ready threads.

A handler thread is not subject to migration under work stealing. Unless the `@(`*proc*`)` annotation is specified, a handler thread becomes a local thread upon creation.

4.11 Migration Threads

Cilk provides a mechanism by which threads may be migrated together with their local data structures during work-stealing. Currently it is implemented as a user level protocol that requires the programmer to specify three threads: the ordinary thread, an additional *migration* (packing) thread, and an unpacking thread.

First, the following statement forms are equivalent to `spawn/spawn_next` except the closure created will not be automatically posted even if it is created full:

`make[_next]_closure` *thr* `(`arg_1`, ...)` *stmt*

Here again, `make_next_closure` is for creating a successor thread, and `make_closure` is for creating a child thread.

The migration thread has the special syntax

`migration thread` *thr* `(`$arg\text{-}decl_1$`, ...)` *body*

Here *thr* must already be defined as a ordinary thread with the same argument prototype. The idea is that at migration time, this thread will be invoked to pack a new closure that is migrated instead. We first explain the protocol using the following example.

Suppose that the thread `foo` will be passed a reference to a locally allocated data structure of some type `typ` (object of type `typ` may further contain pointers, for example to form a linked list):

```
thread foo (cont int k, typ *p) { ... }
```

As a result, `foo` cannot be migrated since `p` will point to garbage on another processor.

The solution is for the user to specify a migration thread to pack the data structure into a new closure, which is migrated instead of the original closure, and to specify an unpack thread which will be used to create a local copy of the data structure once the new closure has reached its destination:

```
migration thread foo (cont int k, typ *p)
{
    closure *cp;
    char *a;

    make_next_closure foo_unpack (k, :a[get_size(p)]) {
        copy_into (a, p);
        cp = $;
    }
    return (cp);
}
```

The way the data structure is copied completely depends on the user supplied function `copy_into`, which for example can flatten a linked list into array `a`.

The new closure must be returned to the scheduler, which posts it to the destination processor instead of the original closure for `foo`. In the trailing statement, the special symbol `$` is bound to a pointer to the newly created closure for this purpose.

In the migration thread, the new closure is made for the thread `foo_unpack`, which must do the reverse of the migration thread:

```
thread foo_unpack (cont int k, char a[..])
{
    typ *p = reconstruct_from (a);
    call_next foo (k, p);
}
```

The unpacking thread uses the user supplied function `reconstruct_from` to make a local copy of the original data structure, then directly calls `foo`.

In order for things to work out correctly, the migration thread must be prototyped along with the original thread and the unpacking thread, and it must have the same argument list as the original thread. The unpacking thread, however, is not related to the original thread, and it is the programmer's responsibility to provide the unpacking thread with the desired behavior. As a final note, both the migration thread and the unpacking thread should use the `_next` variants of the required primitives so that the resulting thread will execute at the same level as the original thread would have executed.

4.12 Shared Memory

We use the following syntax to integrate shared memory into the Cilk language:

- The `shared` keyword identifies pointers to shared memory and arrays allocated in shared memory.[4]
- The `register` keyword on an array or pointer provides fast (but restricted) access to shared memory.

One crucial advantage of this syntax is that a Cilk program written with `shared` pointers can be recompiled easily to run on a system with transparent shared memory, like CilkNOW. In fact, all you have to do is remove the `shared` attribute from every pointer, and your program will run. (The Cilk compiler could do this automatically.)

Section 4.12.1 of this specification presents the user model, explaining the shared memory syntax from a user's perspective. Section 4.12.2 presents some unresolved issues for our implementation of shared memory.

4.12.1 Basic user model

This section describes the three essential new features for the Cilk language: shared memory pointers, register shared memory pointers, and shared memory arrays.

Shared memory pointers

Shared memory is manipulated through shared memory pointers, which behave like C pointers. A shared memory pointer is declared with the type qualifier `shared` on its base type.

[4] Note the similarity to the standard C keyword `const`, which identifies read-only objects in much the same way that `shared` identifies shared-memory objects.

For instance, the following declaration declares `p` as a shared memory pointer with base type `int`:

```
shared int *p;
```

Shared memory pointers behave like ordinary C pointers. The shared pointer type can be used in any context where an ordinary C pointer is legal, including: local and external variables, formal arguments, function return types, struct/union members, and type casts. All operators that apply to ordinary C pointers are overloaded for shared memory pointers, including: indirection (`*`), array reference (`[ ]`), structure reference (`->`), increment/decrement operators, assignment operators, additive operators, relational operators, and equality operators. Two shared memory pointer types are compatible for an operation iff their equivalent C pointer types (after removing the `shared` qualifier) are compatible for that operation. The constant 0 automatically coerces to any shared memory pointer type, producing a null pointer that is guaranteed to be unequal to any pointer to a shared object.

Shared memory pointers are not compatible for assignment with ordinary C pointers. Logically, an object either exists in local processor memory, or it exists in shared memory, never both. Thus it is not possible to convert a local C pointer into a shared memory pointer (see Section reflanref:shmem:usermodel:advanced for an exception to this condition).

An ordinary C pointer may not be assigned to a shared memory pointer, nor vice versa. The sole exception is the constant 0, which may be coerced to a shared memory pointer type even if it has already been coerced to an ordinary C pointer type. This rule allows the standard defined constant `NULL` to serve as a shared memory pointer constant even if it is defined as:

```
#define NULL  (void *)0
```

A shared memory pointer may be explicitly cast into an ordinary pointer, but the results are implementation-dependent. Shared memory pointers may be larger than ordinary C pointers, so some bits may be lost. In any case, the address pointed to by the resulting local pointer has nothing to do with the shared pointer and is almost certainly invalid.

It is legal to cast a shared memory pointer to an integer type and back again, as long as the integer type is large enough to hold a shared memory address.

Register shared memory pointers

The shared memory pointers proposed here are well-suited to *naming* objects in global memory, but less suited to *accessing* objects repeatedly. Each indirection of a shared memory pointer requires a software page-fault check, which is many times the cost of an ordinary memory reference even if no page fault occurs. (The page-fault check costs about 80 cycles in our CM5 implementation.) Frequently, however, the page-fault check is unnecessary, because the page was recently accessed and is still swapped in. Accesses to arrays and structs typically have high spatial or temporal locality, which can be exploited to eliminate many page-fault checks. For repeated access to a shared object (such as an array or structure) with spatial or temporal locality, use a *register pointer*.

A register pointer is declared with the storage class `register`, as in:

```
register shared int *p;
```

The fact that a register pointer has the storage class `register` does not mean that it *must* reside in a machine register, nor does the absence of `register` prevent an ordinary shared memory pointer from occupying a machine register. With respect to allocating `p` to a machine register, the `register` keyword is only a hint which the C compiler is free to follow or ignore.

With respect to shared memory access, however, `register` means something more than register allocation. A register pointer *pins* the local copy of the last shared memory page it read or wrote, so that the page is not flushed by the shared memory manager. Subsequent references to that page through the same pointer are very fast, since only a bounds check is needed to see if the referenced address still lies within the pinned page. (This check consists of a compare and a branch, which take only a few cycles on the CM5.) If the bounds check succeeds — hopefully the common case — then the memory reference proceeds at full speed. If the bounds check fails, then the old page is unpinned, the expensive page-fault check is undertaken, and the new page is pinned as soon as it is available.

Register pointers behave exactly like other shared memory pointers, except for the following restrictions. The first two restrictions are familiar restrictions on the use of `register` in ANSI C. The third is specific to the way register pointers are implemented in Cilk.

- **Only automatic variables and formal parameters may be declared `register`.** The `register` storage class may not be used for external or global variables or struct/union members.

- **It is illegal to take the address of a register pointer.** This rule holds even if the compiler exercises its right to allocate the register pointer on the stack. As a corollary specific to Cilk, **it is illegal to assign the return value of a `spawn` statement to a register pointer.**

- **A register pointer can only access memory less than `CSM_REGISTER_LIMIT` bytes away from its last assignment.** When you assign an address to a register pointer, that address becomes the center of the register pointer's "window." If the register pointer is moved outside its window by incrementing or pointer arithmetic, then its behavior is undefined. To reach memory outside the window, you must move the window by reassigning the register pointer. Note that only direct assignment with `=` sets the window. Assignment operators like `+=` and `-=` do not set the window. The exact size of `CSM_REGISTER_LIMIT` is implementation dependent. (In the CM5 implementation, it is defined as 1 gigabyte minus 1 page.) The need for this restriction is discussed later in this document (see Section 10.3).

Using register pointers involves a tradeoff. Memory references can be as much as an order of magnitude faster for register pointers than for ordinary shared pointers. This gain in the performance of memory access operations is compensated by a loss in the flexibility of memory access operations (the window restriction described above) and in the performance of other operations. For instance, assigning to and from register pointer variables is typically slower, because the assignment involves extra operations for managing the pinned page. For the same reason, register pointers are slower for access patterns without strong spatial or temporal locality, because time is wasted pinning pages which are not used subsequently.

Another disadvantage of register pointer variables is that every register pointer in scope potentially represents a pinned page. Pinning too many pages can cause poor shared memory performance and thrashing. It may even fill up the cache with pinned pages, a fatal error. Fortunately, a Cilk procedure releases all of its pinned pages when it suspends at a `sync` statement. Only the currently active Cilk procedure will have pages pinned, limited to one page per currently active register pointer variable. The danger arises in C functions (or serial versions of Cilk procedures) that use register pointers, which will keep pages pinned as long as they are active. In fact, a recursive C function using register pointers will pin pages proportional to the depth of recursion. Cilk programmers would be wise to avoid deep serial recursion when using register pointers.

Shared memory arrays

A shared memory array is declared with the type qualifier `shared` on its base type. The `shared` declaration may appear only local arrays, not extern or static. Shared arrays may appear in Cilk procedures, Cilk threads, and even C functions. Shared memory arrays may have variable dimension, but only the first dimension (the number of rows) may be variable.

Here are some examples:

```
void f()
{
   shared double b[2000];
   shared struct { int x, y;} c[50][50];
}

cilk void g(int n)
{
   shared int z[n];
   shared double twodee[n][10];
}
```

Note that the declaration "`shared` *T* `x;`", where *T* is a type, is not legal. If you want to allocate a single object of type *T* in shared memory, use the declaration "`shared` *T* `x[1];`". The object can then be referenced via the shared pointer `x`.

A shared array is allocated when control enters its scope and deallocated at the end of its scope. Thus, in the following example, `x1` and `x2` are allocated at procedure entry, but `y1` and `y2` are reallocated before every iteration of the loop.

```
void f(int n)
{
   /* x1 and x2 are allocated here */
   shared int x1[10];
   shared int x2[n];

   while (--n)
   {
     /* y1 and y2 are allocated here */
     shared int y1[10];
     shared int y2[n];

     ...
   }
}
```

It is illegal to jump into a scope containing a shared array. Jumping out of the scope is permitted, however, and the array is deallocated properly.

As in C, the name of a shared memory array implicitly coerces into a shared memory pointer to its first element. The `register` storage class may also be applied to a shared memory array, so that it pins the page most recently accessed through its implicit pointer. For instance, the following code quickly initializes a shared memory array:

```
{
  register shared int a[n];
  int i;

  for (i=0; i<n; ++i)
    a[i] = i;

  ...
}
```

Advanced features

- **Shared-memory** `memcpy`

  Cilk provides three additional memory-copying operations to supplement the standard C function `memcpy`:

  ```
  void memcpy_ss(shared void *to, shared void *from, long len);
       /* copies to Shared memory from Shared memory */

  void memcpy_ls(void *to, shared void *from, long len);
  ```

```
        /* copies to Local memory from Shared memory */

    void memcpy_sl(shared void *to, void *from, long len);
        /* copies to Shared memory from Local memory */
```

Incidentally, these functions are also used by the Cilk compiler to copy structures to and from shared memory.

- **Dynamic stack allocation**

  For allocating stack memory dynamically, the following primitives are available:

```
shared void *allocate_stmem(SharedSizeT size, int alignment);
void free_stmem(shared void *p);
```

  `allocate_stmem` returns a pointer to a block of `size` bytes aligned on a multiple of `alignment` bytes. (`SharedSizeT` is an integral type appropriate to the size of the shared memory address space. In a 64-bit address space, for instance, `SharedSizeT` might be a 64-bit integer, so that a user can allocate large objects.) Memory returned by `allocate_stmem` is automatically freed at the end of the current Cilk procedure.[5] Stack memory may also be deallocated manually with `free_stmem`, which deallocates not only the block to which its argument points, but all stack memory allocated after it as well.

  Shared arrays are also implemented with `allocate_stmem` and `free_stmem`, so the user should be careful of using both allocation methods in the same function. For instance, when a shared array is deallocated at the end of its scope, all stack memory that the user allocated during the array's lifetime is also freed.

- **Explicit page pinning**

  For even less overhead than register pointers, you may pin and unpin shared memory addresses. Pinning a shared-memory location returns a local C pointer, which can be used to access the location repeatedly without any overhead. The local pointer can also be used to access nearby locations, as long as they are on the same shared-memory page. The Cilk programmer is responsible for ensuring that no locations outside the original page are referenced through the local pointer.

  Two C functions are provided for pinning a shared memory location and the page it resides on: `pin_page`, which pins the page for read-only access, and `pin_page_write`, which pins the page for read-write access. The prototypes of these functions are:

```
const void *pin_page(const shared void *p);
      void *pin_page_write(shared void *p);
```

[5]Note that memory allocated by a C function is not automatically freed at the end of the function, and will persist until either its parent Cilk procedure exits or it is explicitly deallocated. This "feature" may be useful for constructing linked data structures on the stack with recursive C functions.

pin_page is faster (and uses less memory in the page cache), so it is the best choice when the local pointer will only be used for reads. When the local pointer may be used for writes, use pin_page_write. It is illegal to write using a pointer returned by pin_page; such writes may be discarded by the runtime system.

The Cilk programmer is responsible for unpinning the shared memory location after using it, using the following C function:

```
void unpin_page(const shared void *p);
```

A page may be pinned multiple times. After the last pin is released, the page may disappear from the cache, making all local pointers to it invalid.

Note the warnings given in section 4.12.1 about poor cache performance when too many pages are pinned. It is wise to unpin a page as soon as you're finished using it.

- **Page aligned arrays**

  In order to force a shared memory array to be allocated on a page boundary, use the page_aligned qualifier in its declaration. For example, the following declaration allocates exactly a page of integers, aligned on a page boundary:

  ```
  shared page_aligned int a[CSM_PAGE_SIZE/sizeof(int)];
  ```

  This feature is useful for programs that use explicit page pinning to achieve good performance. When an array is page_aligned, and the size of its elements is a factor of the page size, then none of its elements will be broken across a page boundary. For instance, an $n \times n$ blocked matrix might be declared in the following way:

  ```
  typedef double Block[CSM_PAGE_SIZE/sizeof(double)];
  shared page_aligned Block matrix[n/BLOCK_SIZE][n/BLOCK_SIZE];
  ```

  Since matrix is page-aligned, and each of its Block elements is exactly one page long, no Block crosses a page boundary. Thus a Block can be manipulated by pinning its page and using entirely local pointer operations.

4.12.2 Open Issues

This section describes several design issues which have not been resolved by this document.

C Libraries

This document does not address the problem of passing shared memory pointers to standard C library functions. The best solution to this problem might be transparent shared memory, which is expected to be available under Cilk-NOW.

Global and static arrays

Global and static shared-memory arrays might be useful. It would be best to allocate these arrays at compile time, so that their addresses are simply constants, but we wouldn't be able to support multiple file compilation without support from the linker.