Cilk™ 1.2 (Version β1) Reference Manual[1]

Robert D. Blumofe Matteo Frigo Michael Halbherr Christopher F. Joerg
Bradley C. Kuszmaul Charles E. Leiserson Phil Lisiecki
Keith H. Randall Andy Shaw Yuli Zhou

February 24, 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™ 1.2 (Version β1), a C language extension and its supporting runtime system intended for developing continuation-passing style multi-threaded programs on CM-5.

Cilk grew out of efforts in implementing a simple scheduling and execution model on top of CM-5's active message layer, and in adapting it to the needs of real life application programs. Pre-release versions of Cilk has been extensively used within the project SCOUT at MIT in developing and porting to CM-5 several major applications, including ray-tracing, protein-folding, computer chess, monte-carlo simulation of heat transfer and radiocity.

The current release is the consolidation of those efforts. However, much of this release should be regarded as experimental in nature, as many issues are yet to be resolved both by further research and experimentation. Therefore, the next release of Cilk is likely to be very different as the system grows into maturity — the current version only marks the starting point towards that direction.

1.1 What is Included in Cilk™ 1.2 (Version β1)

Release Cilk™ 1.2 (Version β1) includes the Cilk runtime system, the Cilk thread preprocessor (cilkpp), a collection of example programs, and various supporting documents all combined into the current volumn: a tutorial for writing and running Cilk programs, the language reference manual, and the specification of the runtime system interface.

The Cilk language provides an abstraction of threads in explicit continuation-passing style, which is first preprocessed into ordinary C code, then compiled by gcc and linked with the Cilk runtime system.

The Cilk runtime system provides the mechanisms for thread communication, synchronization, scheduling as well as primitives callable within Cilk programs. The runtime scheduler implements a generic scheduling policy based on work stealing, which the programmer can complement or completely override via annotations provided in the Cilk language.

1.2 On-going Work and Plans for the Future

For the current release we have concentrated on providing for runtime dynamic scheduling, but have left out another important aspect of parallel programming, namely that of globally shared data structures. The reason for this is mainly that more work is needed in providing a robust shared data abstraction.

However, since the implementation of a shared data abstraction is orthogonal to that of dynamic scheduling, it can be added later without seriously affecting the current structure of the scheduling mechanism.

In the meantime we are also pursuing several closely related projects. Among these there is an implementation of the Cilk runtime system to run on a cluster of work stations on top of TCP/IP, with additional administration facilities necessary in a distributed computing environment. There is also some work under way in porting the Cilk runtime system to SMP's. Finally, we are experimenting with a simple shared data abstraction that provides local data objects with global pointers.

We hope that the release of Cilk™ 1.2 will provide the stimulus and the tools for further work that is required for the next release of a mature Cilk system, which will include the following features:

- A higher level language hiding the explicit continuation-passing style. This means, among other things, that that language will provide the abstraction of functions, and the language processor will automatically generates continuation-passing style threads. The language processor may even become fully integrated with the C front end.
- An improved runtime system. Implementation of the scheduling mechanism will become more robust and refined ...
- Some model of data structures in globally shared memory. It is likely to be more elaborate if object are allowed to interleave on several processors, such as in the case of arrays, when naming becomes a serious issue. However, the evolution of the shared data abstraction will be completely driven by applications.

1.3 The Organization of This Document

Chapter 2 provides a tutorial of how to use Cilk.

Chapter 3 tells you what you need to do to use Cilk on your local system.

Chapter 4 is a language reference manual for the Cilk™ 1.2 language.

Chapter 5 is a specification of the run-time system used by the Cilk™ 1.2 (Version $\beta 1$) compiler.

The appendices include copyright information, man pages, installation instructions, bug reporting instructions, a short writeup of the Cilk team's development methodology, and a list of things that need to be done to the manual.

In particular, Appendix E.4 tells you how to get onto the Cilk mailing lists.

1.4 Introduction Change Log

```
$Log: intro.tex,v $
% Revision 1.9  1994/11/06  02:19:28  randall
% Added AccumulateDoubleWord, SendDoubleArgument and AccumulateDouble to doc.
%
% Revision 1.8  1994/11/03  01:51:05  randall
% Updated everything to 1.2.  Added changes in 1.2 to the change log.
%
```

```
% Revision 1.7  1994/09/28  19:14:16  randall
% changed 1.0 -> 1.1
%
% Revision 1.6  1994/09/06  17:40:22  bradley
% Many small changes from my things-to-do list.
%
% Revision 1.5  1994/09/02  01:31:38  zhou
% Changed tpp to cilkpp everywhere
%
% Revision 1.4  1994/08/29  16:42:19  zhou
% minor corrections
%
% Revision 1.3  1994/08/26  18:01:44  bradley
% Add change log sections.
%
```

Chapter 2

Using Cilk: A Tutorial

Cilk™ 1.2 is a simple C extension to enable the development of multi-threaded programs on parallel machines. Cilk programs run on top of the Cilk runtime system, which supports the scheduling, communication, and synchronization of threads. On the Thinking Machines CM-5, Cilk is implemented

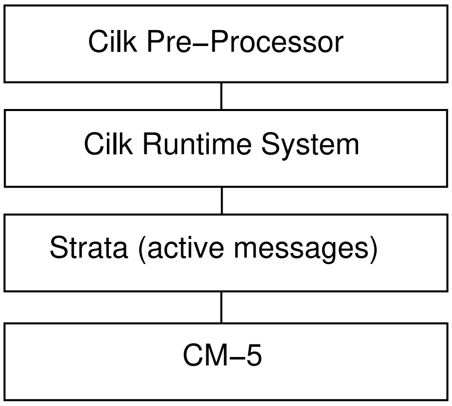

Figure 2.1: Overall structure of the Cilk package on the Thinking Machines' CM-5

on top of Strata, an active message library (see Figure 2.1), although Cilk is relatively machine independent and can be ported to other message-passing or shared memory machines.

2.1 A Simple Example

Figure 2.2 shows a simple C program that computes the Fibonacci function. To demonstrate how Cilk works, we will use as an example the Cilk version of the Fibonacci function shown in Figure 2.3. This program is explained in detail in the following sections.

The program is divided into three sections by "`%%`". The first and the third sections contain normal C code, whereas the middle section (Lines 5–18) contains thread definitions. A thread definition starts with the specifier `thread` and has a parameter list and a body similar to a C function. In the program two threads are defined (Lines 5 and 9).

In order to understand thread semantics, we first need to introduce closures. A *closure* is a data structure that contains a thread pointer (a pointer to the code of the thread) and all arguments needed

```
int fib (int n)
{  if (n<2) return (n);
   else
   {  int x, y;
      x = fib (n-1);
      y = fib (n-2);
      return (x+y);
   }
}

void main (int argc, char *argv[])
{
   int n, result;
   if(argc != 2) {
      printf ("Usage:  %s n\n", argv[0]);
      exit (1);
   }
   n = atoi(argv[1]);
   result = fib (n);
   printf ("Result:  %d\n", result);
}
```

Figure 2.2: Fibonacci function in C

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

%%

thread sum (cont k, int x, int y)
{
   SendWordArgument (k, x+y);
}

thread fib (cont k, int n)
{
   if (n<2) SendWordArgument (k, n);
   else
   {  cont x, y;
      spawn_next sum (k, ?x, ?y);
      spawn fib (x, n-1);
      spawn fib (y, n-2);
   }
}

%%

void main (int argc, char *argv[])
{
   int n, result;
   if(argc != 2) {
      printf ("Usage:  %s n\n", argv[0]);
      exit (1);
   }
   CilkInit();
   cilk_active_size = PartitionSize;
   n = atoi(argv[1]);
   result = RunScheduler (CILK_AUTO, fib, 1, n);
   if (Self == 0) printf ("Result:  %d\n", result);
   CilkExit(0);
}
```

Figure 2.3: Fibonacci function in two threads

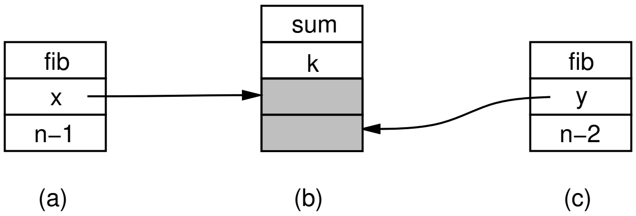

Figure 2.4: Closures created in the fib thread

to run the thread. Closures are created via the spawn or spawn_next statements. For example, Line 15 creates the closure shown in Figure 2.4 (a), which captures the thread pointer fib and the available arguments x and $n - 1$. Because all of the arguments are available to run the thread, this closure is called a *ready* closure. It is immediately handed to the scheduler to be executed either locally or on a remote processor.

Closures with some empty argument slots are called *waiting* closures. Line 14 creates such a closure, as shown in Figure 2.4 (b). This closure is waiting for two of its arguments to be filled in. The forms ?x and ?y allocate empty slots in the closure and at the same time initialize x and y as *continuations* pointing to these empty slots. Continuations are pointers into closure structures and are used by the program to later fill in these empty slots. The statement

```
SendWordArgument (k, n);
```

sends the number n to the empty argument slot referenced by continuation k. The argument slot is filled in with this value, and if all argument values have arrived then the closure is ready and is given to the scheduler.

Closures are the basic mechanism that enables the continuation-passing style of execution. In the Fibonacci program, the thread fib (Line 9) has a argument of type cont (continuation), pointing to a slot where the return value should be sent. If $n \geq 2$, it creates three closures: a waiting closure (Line 14) with two empty slots x and y, and two full closures (Lines 15 and 16), each one assigned to compute a value to put in one of these two slots. The full closures are immediately given to the scheduler which will execute them either locally or on another processor. These computations correspond to two recursive calls to the Fibonacci function with arguments $n - 1$ and $n - 2$. These routines compute their results and send them to the waiting closure. On receiving its arguments, the waiting closure becomes ready and it is given to the scheduler. The scheduler decides where and when this closure will be executed. When executed, this closure will return the sum of its arguments to the continuation k.

It is instructional at this point to compare the multi-threaded program to the typical C function definition for Fibonacci such as the one in Figure 2.2. We note the following properties of a thread in continuation-passing style that make it different from the procedure-invocation style of the C code:

- Threads do not return values. Instead, continuations are explicitly passed among threads and the threads send values to each other using these continuations.

- Threads are non-blocking. Instead of waiting for recursive calls to return their results as is the case in the sequential program, the waiting part is lifted as a different thread (the sum thread). The parent thread just spawns new threads and continues to the end, at which point it relinquishes control to the scheduler.

Because threads are non-blocking, we say that they are *split-phase* implementations of procedures. Split-phase means that the spawning thread does not wait around to gather the results of its spawned children itself. Instead, it creates a new thread (in Fibonacci, the `sum` thread) to gather the results for it. Looking at it this way, there is a natural grouping of threads into the abstract procedure invocations that they implement. For instance, the fib thread and its associated `sum` thread group together to form one invocation of the regular C fib procedure.

Because the scheduler can use information about which threads are part of the same split-phase operation (i.e. which threads comprise one procedure invocation), two `spawn` primitives are provided. The primitive `spawn_next` should be used when the spawned thread is part of the same procedure invocation as the thread which spawned it. The primitive `spawn` should be used for the threads that represent procedure calls from within the spawner's procedure invocation (i.e., the children of the spawner's procedure invocation). A good rule of thumb is that all closures that are created full should be `spawn` threads, and all waiting closures (i.e., all closures with a `?` in them) are `spawn_next` threads.

To finish describing the fib program, the primitives on line 27 and line 32 initialize and stop the Cilk system. The argument for `CilkExit` is an exit error code that functions in the same manner as the exit error code for `exit()`. Also, the primitive

```
RunScheduler (CILK_AUTO, fib, 1, n);
```

provides the entry into the scheduler. Here one specifies the scheduling mode (`CILK_AUTO` means automatic thread migration via work-stealing, see section 5.4 for other modes), the first thread to run (fib), the number of arguments supplied to fib (1) and the actual argument (`n`). The actual definition of the fib thread specifies an additional first argument which is the continuation to which fib sends the result of its computation to, and this continuation will be supplied by the scheduler.

2.2 Running the Example

There are three steps to running the Fibonacci program on your target machine. The source code can be found under the name `fib.p` in the `examples` directory of the Cilk distribution. First, copy `fib.p`, `Makefile`, and `job` into your working directory from the `examples` directory. Make sure the `STRATA_DIR` and `CILK_DIR` assignments in the Makefile point to each of their distribution directories, respectively (see chapter 3 for the locations of these directories). Then type `make fib_cm5_st` and the executable will be made for you. The `_cm5` extension is used to denote files that are used on the `CM5`, and the `_st` extension is for files with statistics gathering enabled. To run fib, just type `jrun job` and the `fib_cm5_st` executable will be run on the `CM5`.

The Makefile includes the following steps:

1. Run the pre-processor (`cilkpp`) on `fib.p` to generate `fib.c`, its C-code equivalent.

2. Compile `fib.c` using `gcc`.

3. Link the object file with one of the Cilk libraries in the distribution directory.

All of these tasks are performed automatically by the Makefile provided in the `examples` directory. The user is strongly encouraged to use this Makefile because it correctly sets compile-time flags and links in the correct libraries for a proper executable.

Note: Certain libraries have been compiled with certain compile-time flags, and any user code linked with these libraries MUST have been compiled with the same flags. Using the Makefile is an easy way to ensure a correct match.

To run other examples in the `examples` directory, just copy their `.p` files into your working directory, make them, and edit the `job` file to uncomment the examples you want to run. Then type `jrun job` to run them.

Note: Some of these instructions are specific to the CM5. Check chapter 3 for the local guide to running this software.

2.3 Cilk Internals

The Cilk scheduler is a SPMD program that is run on all the processors. The scheduler manages a queue of full threads on each processor. For local execution, the scheduler tries to execute threads which are farther down in the procedure activation tree. This heuristic is (we believe) good for keeping space usage at a minimum.

The scheduler also attempts to load-balance the machine using a work-stealing algorithm. When work is stolen, the scheduler tries to migrate closures higher up in the procedure invocation tree. This strategy is used because closures higher in the invocation tree usually represent larger pieces of work.

2.4 Why Continuation-Passing Style

Since threads are obviously more difficult to write than ordinary C functions extended with `fork`'s and `wait`'s, one may well question what is to be gained by adopting threads in such explicit continuation-passing style. In addition to programming style, people familiar with programming language implementation may object to the way function frames are broken up into closures for threads, which requires more frequent copying of arguments.

The only answer to these questions lies in the simplicity of these threads and their execution model. This very simplicity translates into a simple and clean implementation of Cilk. For example, the part of the Cilk runtime system on the CM-5 to support the threads abstraction, including the scheduler and other primitives, is only 2000 lines of C code. The Cilk syntax extension requires no more than macro expansions, and the resulting C code is well structured and very readable (it is not, as might be imagined, assembly code written in C).

Since Cilk threads are at a lower level than the functional abstraction, it would be possible and beneficial to cast alternative implementations as either embelishment (such as a higher level language) or optimizations (such as packing closures belonging to the same function invocation into a frame) of the basic model, gaining convenience of expression and/or execution efficiency at the cost of more complexity.

2.5 More Advanced Features

2.5.1 Calls and Tail Calls

Cilk provides several primitives that can be used in place of `spawn` that may improve your application's performance. The first of these is the `call` primitive. The `call` primitive simply does a normal, C-style function call to the named thread. This is useful for speeding up your application because it avoids the overhead of the scheduler. However, using this primitive too much may restrict the available parallelism in your application. A safe rule of thumb is to only convert the last `spawn` in your thread body into a `call`.

Because the thread you are calling is executed immediately, all arguments must be present (i.e., no `?x` declarations).

A `call` primitive is easily substituted into the fib thread, as shown below:

```
thread fib (cont k, int n)
{
   if (n<2) SendWordArgument (k, n);
   else
   {  cont x, y;
      spawn_next sum (k, ?x, ?y);
      spawn fib (x, n-1);
      call fib (y, n-2);
   }
}
```

The second new primitive is the `tail_call` primitive. This primitive is an optimized version of the `call` primitive that can be used instead of the `call` primitive when the following three conditions are satisfied. First, the thread being called must be the same as the thread the `tail_call` appears in. Second, the `tail_call` primitive *must* be the last statement in the thread (this is because `tail_call` never returns to its caller). Finally, all local variables and arguments must be dead. In particular, you can't have pointers referencing any of the data in the C stack frame. This primitive is extremely fast because it expands into one assignment for each argument and a jump to the beginning of the thread code.

The code for fib satisfies all the requirements listed above and therefore the last `spawn` can also be a `tail_call`.

Finally, just as there are `spawn` and `spawn_next` primitives, there are also `call_next` and `tail_call_next` primitives.

Note: There is one further distinction: Arrays are passed by *reference* when using the `call` and `tail_call` primitives (whereas arrays are passed by *value* when using the `spawn` primitives). This is usually not a problem because only the last routine in a thread is one of these optimized spawns. However, this distinction can lead to problems so use these primitives carefully.

2.5.2 Accumulators and Signals

Accumulators can be used to gather results from several threads into one argument slot. To use an accumulator, you first create an accumulator slot using the following syntax:

```
spawn foo (?k{n:init});
```

This code says that `foo` has one argument which is an accumulator slot. This slot will accumulate `n` values with an initial value of `init`. This code also initializes `k` as a continuation pointer to this slot. To accumulate values into this slot, we use the `AccumulateWord` primitive:

```
AccumulateWord (k, accum_word_add, val);
```

The `AccumulateWord` primitive accumulates `val` into the accumulator slot pointed to by `k` using, in this case, the operator `accum_word_add`. Thus when `foo` is run, its argument will contain the sum of all `n` values that were sent to it.

Here is an example of how to use accumulators in the Fibonacci program:

```
thread sum (cont k, int n)
{
   AccumulateWord(k, accum_word_add, n);
}

thread fib (cont k, int n)
{
   if (n<2) AccumulateWord (k, accum_word_add, n);
   else
   {  cont x;
      spawn_next sum (k, ?x{2:0});
      spawn fib (x, n-1);
      tail_call fib (x, n-2);
   }
}
```

There are several accumulator operations defined for you. You can also define your own accumulator operations if you wish. See section 5.4 for details.

Signals are just accumulators without any values. These are useful when you want to know when a group of threads have completed. The syntax for creating a signal slot is the same as for an accumulator slot except that there is no initial value:

```
spawn foo (?k{n});
```

Also, in the prototype for `foo`, the argument slot for the signal must be of type `signal`. To send a signal to a continuation, just call

```
Signal (k);
```

Here is an example of how you might use signals to create a barrier in your code. The thread `spawner` runs several `task` threads, and each of these signals the `barrier` thread when it is done its work. The `barrier` thread will then send a value on to continuation `k` when all of the signals have arrived.

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

thread task (cont s)
{
   ...  do some work ...
   Signal(s);
}

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

2.5.3 Explicit Posting of Threads

It is often useful to be able to post certain threads on specific processors. For instance, if there is a data structure that is statically laid out on your machine and you want to post threads to where the data they will use resides, you will need to use this mechanism. Also, you can use this mechanism to explicitly schedule your computation (this is done in conjunction with using CILK_MANUAL mode, see section 5.4). The statement

```
spawn fib (k, n)@pn;
```

posts the thread `fib (k, n)` on the processor numbered `pn`. If the thread is ready to execute, it will be posted there immediately, and if it is unfilled it will be posted there when it becomes filled. 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 work (k)@i;
}
```

The pre-defined variable `Self` can be used to force a closure to be executed locally.

```
spawn fib (k, n)@Self;
```

Note: the keyword `local` used in a thread definition will force any spawns of that thread to be executed locally.

2.5.4 Arrays as Closure Arguments

In standard C, arrays must be passed by reference (i.e., by pointer). However, in order to facilitate migrating threads, Cilk passes arrays by value (except in `call` and `tail_call` primitives: see section 2.5.1). For example, the following line of code defines a thread that takes both a fixed size and a variable size array as arguments:

```
thread foo (int a[10], int b[]);
```

The array `a` is a fixed-length array of length 10, and array `b` is a variable-length array whose size will be determined when the closure for `foo` is created.

Note: Variable-length arrays are only allowed in the last argument slot.

When you want to spawn thread `foo`, there are several different ways to specify the values of the arrays to be passed. The first method is to copy the array argument from another array. The form

```
spawn foo (x[..], y[2..8] )
```

can be used to copy 10 elements of array `x` into the array `a` and 7 elements of the array `y` (from indices 2 to 8, inclusive) into the array b in the closure for `foo`. Note that fixed arrays do not need range specifiers because the size is set in the thread declaration for `foo`.

The second method to initialize arrays is to initialize them as an array of slots that another thread will fill in using `SendArgument` calls. This is done using the following syntax:

```
cont x,y;
int i;
spawn_next foo (?x[..], ?y[7]);
   for (i=0; i<10; i++)
      spawn bar (IndexContinuation(x,i,sizeof(int)));
   for (i=0; i<7; i++)
      spawn bar (IndexContinuation(y,i,sizeof(int)));
```

Space for `a` and `b` are allocated just as before, but instead of getting pointers to these arrays and filling them in immediately, we get continuations pointing to these arrays and we fill them in using `SendArgument` calls (in this case, the `SendArgument` calls are hidden inside the thread `bar`). The function `IndexContinuation(x,i,sizeof(int))` gives the continuation for the `i`th element of the array of `int`s pointed to by continuation `x`.

The last method to initialize arrays is to use *initialization pointers*. This is done as follows:

```
int *x,*y;
spawn foo (:x[..], :y[7])
   {
      memcpy(x, xdata, 10*sizeof(int));
      memcpy(y, ydata, 7*sizeof(int));
   }
```

The form `:x[..]` is used for fixed-size arrays and initializes `x` to point to the uninitialized array `a` in the closure of `foo`. The form `:y[7]` is used for variable-size arrays and both allocates an array of size

7 for b in the closure for foo and initializes y to point to that array. The user then can fill in the arrays as he or she pleases in the trailing statement to the spawn (any spawn primitive can have a trailing statement, possibly compound, that is executed after the closure is created but before it is posted. This statement is useful for initializing certain parts of the closure, in this case array values).

Note: initialization pointers are only valid inside the spawn's trailing statement.

2.5.5 Packing/Unpacking Closures for Migration

In the examples you have seen so far, all arguments were passed by value in the closures. Closures of this type are called *flat* closures because there are no pointers to heap objects or other structures. Flat closures are easy to migrate because there are no dependencies on memory locations other than the closure itself.

Non-flat closures, however, are more difficult to migrate because heap objects pointed to by the closure need to be packed up and sent to the destination node along with the closure itself. Because this pointer structure can be arbitrarily complicated, Cilk provides a mechanism for users to specify how they want their closures to be packed for migration. For each thread you define, you may also define a *migration* thread whose job it is to pack a non-flat closure into a flat closure for migration. Here is an example of how to use a migration thread:

```
thread foo (int *array)
{
   ...  code for foo ...
}
migration thread foo (int *array)
{  closure *cp;
   make_next_closure foo_unpack (array[..])
      { cp = $; }
   free (array);
   return (cp);
}
thread foo_unpack (int array[10])
{ int *new;
   new = malloc (10*sizeof(int));
   memcpy (new, array, 10*sizeof(int));
   call_next foo (new);
}
```

Here, foo is our typical thread with a non-flat closure. In order to migrate thread foo, we need to define two new threads. The first is the packing thread, defined using the keyword migration. The prototype for this thread must be exactly the same as the prototype for the thread that is to be migrated. When the scheduler decides to migrate a thread with a corresponding migration thread, the migration thread is called with the arguments of the closure to be migrated and it is expected to return a flat closure that will then be sent to the destination node. This flat closure should have as its thread pointer the code for the unpacking thread to be run on the destination node.

In the example above, the migration thread for `foo` makes a closure for `foo_unpack`, with the argument for `foo_unpack` copied from the `array` argument from `foo`. The expression in brackets is used to obtain a pointer to the newly allocated closure (in a trailing statement to `spawn` or `make_closure`, the special symbol `$` refers to the closure just created). The migration thread then deallocates the storage used by `foo`'s pointer structures and returns the flat closure.

On the destination node, the thread `foo_unpack` allocates a new array `new` and copies the transferred array into it, and then does a `call_next` to the original `foo` thread. In this whole process, please note the following items:

- During migration, all primitives used are of the `_next` variety. This is done to ensure that the resulting migrated thread is at the same level as the original thread (see 5.4 for a discussion of levels).
- It is the responsibility of the unpack thread to make sure that the thread it unpacks is executed before it is migrated again. An easy way to do this is to use the `call_next` primitive to run it right away. An alternative is to use the `spawn_next` primitive with the `@Self` directive.
- Using the `@n` directive after a `spawn` primitive overrides the migration mechanism and sends the closure directly, even if it has a migration thread. Therefore, you must manually pack any thread that you wish to explicitly post to a particular processor.

2.5.6 SendArgument variants

There are several different flavors of the `SendArgument` routine for sending different size arguments. You have seen the `SendWordArgument` routine in previous examples. Here are the other routines for passing arguments to continuations:

```
SendCharArgument (cont k, char c);
SendShortArgument (cont k, short s);
SendWordArgument (cont k, Word w);
SendDoubleWordArgument (cont k, DoubleWord d);
SendFloatArgument (cont k, float w);
SendDoubleArgument (cont k, double w);
SendArrayArgument (cont k, char *array, int length);
```

These routines send their argument to the continuation slot of the corresponding size pointed to by the continuation `k`. For `SendArrayArgument`, the size of the destination slot is `length` bytes long. `SendArrayArgument` is useful for sending both arrays and structures.

Caution: Continuations are not typed, so care must be used to send the right size argument to continuations that point to a certain size slot. Thus, a continuation for a `char` slot should not be used in a `SendWordArgument` call. Violating this rule will cause your program to crash. Note that we expect to type continuations in the next version of Cilk.

2.5.7 Global Data Structures

Global pointers are provided for maintaining distributed data structures and other distributed applications. Global pointers consist of a processor number and a local (regular) pointer into that processor's memory. Global pointers are created using the `MAKEGLOBPTR` macro which takes a local pointer and produces a global pointer. The processor number and local pointer of a global pointer can be obtained using the `PN` and `OFFSET` macros. There is an example of using these macros in the Cilk examples directory under the name `tree.p`.

2.6 Tutorial Change Log

```
$Log: tc-tut.tex,v $
% Revision 1.13  1994/11/06  02:19:49  randall
% Added AccumulateDoubleWord, SendDoubleArgument and AccumulateDouble to doc.
%
% Revision 1.12  1994/11/03  00:35:51  randall
% Added SendFloatArgument and AccumulateFloat to manual.
%
% Revision 1.11  1994/09/28  18:21:21  zhou
% Modify examples to version 1.1 syntax
%
% Revision 1.10  1994/09/06  17:40:30  bradley
% Many small changes from my things-to-do list.
%
% Revision 1.9  1994/09/02  19:22:36  randall
% StrataInit -> CilkInit, same for exit.
%
% Revision 1.8  1994/09/02  01:31:45  zhou
% Changed tpp to cilkpp everywhere
%
% Revision 1.7  1994/08/29  21:48:20  randall
% Put serial fib ahead of threaded fib in tutorial.
% Changed some local guide stuff and how to obtain Cilk stuff.
%
% Revision 1.6  1994/08/26  19:22:42  randall
% Added change log to the tutorial.
% Did some editing of the language reference.
%
```

Chapter 3

Local Guide

This local guide explains how to use Cilk™ at MIT. If you are not at MIT, you probably need slightly different instructions.

A copy of Cilk can be found in `/a/randall/Cilk1.2/` on `scout.lcs.mit.edu`. There is an example `Makefile` that you can use to build your own projects in `/a/randall/Cilk1.2/examples/`.

In order to get `flex` (a lexical analyzer) added to your path, type
'`source /usr/local/conf/sys-dots/std.cshrc`' on `scout`. You will need this for compiling cilkpp.

An on-line version of this manual can be found from CSG hosts as

`file://localhost/home/prj/Cilk/unreleased/doc/manual/manual.html`

To access `/home/prj/` from a TOC machine, use `/csg/prj/`. For example,

`file://localhost/csg/prj/Cilk/unreleased/doc/manual/manual.html`

Chapter 4

Cilk Language Reference Manual

This chapter describes the Cilk™ 1.2 language extension to C that enables multi-threaded programming on parallel architectures. The extension provides an abstraction of threads in explicit continuation-passing style, which is preprocessed to ordinary C code, then compiled to run on top of the Cilk runtime system.

The Cilk runtime system implements a scheduler that by default uses work stealing for dynamic load balancing in order to maximize computation locality. The programmer can also gain control over scheduling via a system of annotations, both with or without work stealing in effect.

In the remainder of the section we shall first define the key underlying runtime concepts, then proceed to describe the details of the language.

4.1 Basic Concepts

In Cilk, the basic unit of scheduling and execution is a *thread*. Threads are defined with a syntax similar to C functions. For example, the following are two threads that computes the nth Fibonacci number:

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

thread fib (cont k, int n)
{
    if (n<2) SendWordArgument (k, n);
    else
    {   cont x, y;
        spawn_next sum (k, ?x, ?y);
        spawn fib (x, n-1);
        spawn fib (y, n-2);
    }
}
```

Conceptually, threads form an abstraction that is more primitive than functions. As can be seen from the above example, the code that belongs naturally to a function is split into more than one threads.

However, the notion of a function is still visible in Cilk. In the above example, `spawn_next` is used to create a thread that "continues" within the same function, while `spawn` is used to create a thread corresponding to a function invocation.

At runtime, a thread is invoked from, and returns to the scheduler. Inter-thread communication uses explicit continuation-passing style to send arguments and results. Intuitively, a thread groups together a sequence of instructions without synchronization events (i.e., waiting for something to occur) in the middle of its execution. This arrangement results in a very flexible and simple scheduling and execution model.

4.1.1 Closures

Normally, a thread is created by making a closure. A closure is just a data structure that captures all information needed to execute the thread. More specifically, the statement

`spawn` *tp* `(`arg_1`, ...);`

creates a closure containing the thread pointer *tp* and one slot for each argument (the actual implementation may need to keep additional information):

tp
arg_1
$\vdots$

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*.

4.1.2 Continuations

A *continuation*[1] is a global reference to an empty argument slot, to which an argument can be sent. For example, the statement

```
spawn_next sum (k, ?x, ?y);
```

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

	`sum`
	`k`
`x` $\rightarrow$	
`y` $\rightarrow$	

Usually, both `x` and `y` will be passed to new threads which will eventually send arguments via some runtime primitive such as

`SendWordArgument (x,` *arg*`);`

that fills the slot referenced by `x`. The closure will be *posted* when it becomes full, and available for execution via the scheduler.

[1] Note that we are somewhat abusing the term "continuation" here, which normally means "the rest of the computation".

Continuations can be created to reference argument slots of different types, and different argument sending primitives must be used to match the size and the semantics of the argument. This will be further described in section 4.9.3.

4.1.3 Thread Scheduling and Work Stealing

During the execution of a Cilk program, thread spawning forms a proper tree, with data-dependencies among threads (i.e., those indicated by continuations) forming a DAG. In a correct program, the order in which ready threads are executed should not change its result, but may have drastic effects on its space and time efficiency. Therefore in order to write a good Cilk program it is necessary to understand Cilk's basic scheduling and execution model.

At runtime, the Cilk scheduler groups together closures corresponding to function invocations (i.e., the closure created by `spawn` and those created by subsequent `spawn_next`'s). It orders these groups by the time of the first `spawn`, and within each group the closures are ordered by the time when they are posted. For local execution, the scheduler always tries to take the most recently posted closure in the most recently invoked function.

A work stealing mechanism is implemented so that each processor sends a request to a randomly chosen target processor when it runs out of its local pool of ready threads. The target processor, upon receiving the request, will try to migrate the most recently posted closure belonging to the oldest function invocation, or reply with a negative response to the requesting processor. In the latter case, the requesting processor will have to repeat the work-stealing cycle for as long as its local pool of ready threads is empty.

Work stealing has proven to be extremely effective in reducing the frequency of thread migrations required by load balancing, at the same time creating computation locality which is critical for keeping performance close to that of a sequential program.

For more complicated situations, Cilk also allows the programmer to specify where, and to some extent when, a thread must be executed. This can be done either with or without work stealing in effect. In this case the programmer will assume part or all of the responsibilities of load balancing, preserving computation locality and limiting resource consumption.

4.2 Program Structure

A Cilk program consists of three sections separated by `%%`:

program: *C code*
`%%`
$thread_1$`;`
$\vdots$
`%%`
C code

The program must be first preprocessed to ordinary C code, and then compiled and linked with the runtime library.

Since the preprocessor copies both of the C code sections, the C syntax there will be whatever is accepted by the C compiler being used[2].

The thread definition and declarations in the middle section are expanded by the preprocessor, which accepts a mostly C syntax with some differences. Some of the differences are due to Cilk extensions, others are due to implementation constraints. In the following sections we shall describe in detail only the Cilk specific part of the language. One should keep in mind, however, that any limitations or changes to C syntax within the threads section do not apply to the C code sections.

4.3 Keywords, Constants and Operators

All C keywords used in Cilk thread definitions and declarations will appear in the constructs described in later sections. In addition Cilk introduces the following keywords:

`closure`, `cont`, `globptr`, `signal`, `type`
`handler`, `local`, `migration`, `thread`
`make_closure`, `make_next_closure`, `spawn`, `spawn_next`,
`call`, `call_next`, `tail_call`, `tail_call_next`

Constants can be specified in the following format: all C character and string constants, integer constants such as `12`, `0711` (octal) and `0xFA` (hexadecimal), and float constant such as `1.27` or `.34`.

The following operators can be used in the body of thread definitions (listed in increasing order of precedence):

Operators	Associativity
`@`	none
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`	right to left
`?`, `:`	right to left
`||`	left to right
`&&`	left to right
`|`, `^`	left to right
`&`	left to right
`==`, `!=`, `<`, `<=`, `>`, `>=`	left to right
`<<`, `>>`	left to right
`+`, `-`	left to right
`*`, `/`, `%`	left to right
`!`, `~`, `++`, `--`, `sizeof`	right to left
`( )`, `[ ]`, `.`, `->`	left to right

All operators except `@` are inherited from C with their original meaning.

4.4 Thread Definition and Declaration

Similar to a C function definition, a thread definition has the form

thread: [*attrib*] `thread` *tp* `(`*arg-decl*$_1$, ... `)` *body*

where *body* is a compound C-like statement enclosed in braces. The body can also be the null statement

[2] For Cilk™ 1.2, `gcc` is needed as the runtime system and the preprocessor output rely on certain `gcc` extensions.

(`;`), in which case it is a thread declaration. A thread declaration or definition must precede its first use, thus declarations are necessary in cases where threads are defined recursively, or where threads defined in separate files are cross-referenced.

The optional *attrib* can be one of

`local`, `handler`, `migration`,

Their use will be described later in section 4.5.2 and 4.6.

4.4.1 Types of Thread Arguments

A thread argument declaration has the following syntax:

arg-decl:	*type var-ref*	
	`signal` [*var*]	
type:	`char` | `short` | `int` | `long`	
	`float` | `double`	
	`closure` | `cont` | `globptr`	Cilk specific
	`struct` *var*	
	`type` *var*	
var-ref:	*var-decl*	
	`*`*var-ref*	
var-decl:	*var*	
	var`[`*expr*`]`	fix-size array
	var`[]`	variable-size array

The declaration syntax here is mostly a proper subset of the C syntax, with some notable differences.

`closure`, `cont`, `globptr` and `signal` are types specific to Cilk.

Pointer and structure types follow the C convention.

Array types differ from the C convention. Due to C's integration of arrays and pointers, a pointer is always passed for an array argument. In Cilk an array is normally passed by value (see section 4.5.1). In addition, *var*`[`*expr*`]` specifies a fixed-size array where *expr* is the constant size expression, and *var*`[]` specifies a variable-size array whose size will be provided at thread creation time. A variable-size array is only allowed as the last argument of a thread.

The form `type` *var* provides a back door to the C type definition mechanism, here *var* can be any variable defined using `typedef`. The keyword `type` here is needed since Cilk does not understand types defined outside the threads section, thus

```
foo *bar;
```

would be ambiguous as to whether it is a declaration or an expression used as a statement.

4.4.2 Body of Thread Definitions

Thread body has the form of a C compound statement:

body: $\{\ decl_1 \ldots stmt_1 \ldots \}$

Here the declarations follow the C convention, except with the same limited syntax as thread argument declarations. However, simultaneous declarations such as

```
int i, *p;
cont x, y;
```

are allowed. Almost all of the C statement and expression syntax are allowed, and we list them here for easy reference:

stmt:	`;`	the empty statement
	expr`;`	
	label`:` *stmt*	
	`case` *expr*`:` *stmt*	
	`default:` *stmt*	
	`goto` *label*`;`	
	`break;`	
	`return` [*expr*]`;`	
	`if (`*expr*`)` *stmt* [`else` *stmt*]	
	`switch (`*expr*`)` *stmt*	
	`for (`[*expr*]`;` [*expr*]`;` [*expr*]`)` *stmt*	
	`while (`*expr*`)` *stmt*	
	`do` *stmt* `while (`*expr*`);`	
	$\{\ decl_1 \ldots stmt_1 \ldots \}$	
expr:	*cnst*	
	expr`[`*expr*`]`	
	expr`.`*var*	
	expr`->`*var*	
	`(`*type*`)` *expr*	
	uop expr | *expr uop*	See section 4.3 for the list of operators
	expr bop expr	
	expr`?` *expr*`:` *expr*	
	expr `(`$expr_1, \ldots$`)`	
	`sizeof (`*type*`)`	
	`(`*expr*`)`	

4.5 Thread Creation

In addition to ordinary C statements, the following Cilk statement is used to create a new thread by making a closure:

`spawn[_next]` *tp* `(` $arg_1, \ldots$ `)` [`@`*expr*] *stmt*

here *tp* must be defined or declared as a thread with the correct number and types of arguments. The keyword `spawn` should be used to create a thread that starts a new function invocation, whereas `spawn_next` should be used to create a thread that is logically "continuing" execution within the same function. Their usage may affect the performance of the program, and especially that of work stealing (see section 4.1.3).

If the optional `@`*expr* is present, the closure will be posted to the processor with number equal to *expr*, otherwise it will be posted locally. If all arguments are available then posting is immediate.

The trailing *stmt* can be null (`;`) or any C statement. It is executed before the closure is posted, useful for further initializing the closure. For this purpose, within the *stmt* the special symbol `$` is bound to a pointer to the newly created closure. It can be accessed, for example, as in

```
closure *cp;
...
spawn foo (...) { ...; cp = $; ... }
```

4.5.1 Thread Arguments

Much of the syntax sophistication in Cilk lies in different ways of specifying the arguments for thread creation. Thread arguments differ from ordinary C function arguments:

arg :	*expr*	ordinary C expression
	expr`[..]`	array argument
	expr`[`$expr_l$`..`$expr_u$`]`	variable-size array argument
	`?`*var*	inlet
	`?`*var*$\{expr_j$`:` $expr_i\}$	accumulator inlet
	`?`*var*$\{expr_j\}$	signal inlet
	`?`*var*`[..]` | `?`*var*`[`*expr*`]`	array of inlets
	`:`*var*`[..]` | `:`*var*`[`*expr*`]`	initialization pointer

A thread argument of the form

expr

is just a normal C expression.

An argument of either 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 be a continuation pointing to that slot. In all of the forms *var* must be declared to have the type `cont`. The size of the slot is determined from the corresponding type declaration of the thread argument in the thread prototype.

The argument forms here differ in how many arguments are expected and in the way the argument is deposited in the empty slot. More specifically:

- `?`*var* allocates a *inlet* slot that waits for exactly one argument to be sent.
- `?`*var*$\{expr_j$`:` $expr_i\}$ allocates an *accumulator inlet* slot, expecting $expr_j$ values and initializes the slot to $expr_i$. The closure is not considered full until all $expr_j$ arguments arrives. How the arguments are to be accumulated is completely unspecified and depends on which runtime primitives are used to send the arguments.
- `?`*var*$\{expr_j\}$ specifies a *signal inlet*. 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 inlet). As in the previous case, the closure will be waiting for an additional $expr_j$ signals.

Depending on the slot type, different Cilk runtime primitives need to be used for sending arguments, as described further in section 4.9.3.

An array argument has one of the following forms

expr`[..]`
expr`[`$expr_l$`..`$expr_u$`]`

and stores the array *expr*`[`*size*`]` in the closure. An argument of the first form must be declared to have an array type of fixed-size *size*. An argument of the second form must have a variable-size array type, and the array segment includes elements at both the lower and upper bounds (*size* is calculated as $expr_u - expr_l + 1$).

Both of the following argument forms

`?`*var*`[..]`
`?`*var*`[`*expr*`]`

allocate an array of inlets and initializes *var* as a continuation pointing to the beginning of the array. The first argument form must have fixed-size array type, and the size is taken from the type declaration. The second form must have variable-size array type. Several runtime primitives can be used to send the array as a whole, in pieces or element-wise (see section 4.9.3).

Arguments of the following forms

`:`*var*`[..]`
`:`*var*`[`*expr*`]`

are called *initialization pointers* and must have corresponding array types (the first form must have fixed-array type, with the size taken from the type declaration. The second form must have variable-size array type). Either form allocates an array of the specified size in the closure, and sets *var* to be a pointer to the beginning of the array. Unlike array arguments, the array allocated here is uninitialized. It is the programmer's responsibility to properly initialize the array using the pointer *var*. In addition, *var* must be declared to match the array element type. Since full closures are automatically posted, the initialization pointer should not be referenced outside of the trailing statement to the thread creation.

4.5.2 Annotations for Thread Scheduling

Under the default Cilk scheduling policy, the programmer does not specify either where or 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 function 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` *tp* `(`*arg-decl*$_1$`, ...)` *body*

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

`spawn[_next]` *tp* `(`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` *tp* `(`*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.

4.5.3 Direct and Tail Calls

In addition to spawning threads, the following direct and tail call forms can also be used:

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

`call` invokes the thread *tp* as a C function. `tail_call` is more efficient, but is restricted in that *tp* 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 as in `spawn_next`, i.e. it tells the runtime system that the thread should be considered as logically continuing within the same function.

Since there is no closure involved, arguments in a direct or tail call form must only be C arguments and must be specified in full. Array arguments can only be passed by reference to direct or tail called threads. Since a normally scheduled thread keeps its closure until the end of execution, such array references should remain valid within a direct or tail called thread.

4.6 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` *tp* `(`arg_1`, ...)` *stmt*

Here again, `make_next_closure` is for creating a thread within the same function, `make_closure` is for creating a thread corresponding to a new functional invocation.

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

The migration thread has the special syntax

`migration thread` *tp* (*arg-decl*$_1$, ...) *body*

Here *tp* 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 `tp` (object of type `tp` may further contain pointers, for example to form a linked list):

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

    make_next_closure foo_unpack (k, :a[size]) {
        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 (recall from section 4.5 that `$` is bound to the pointer to the newly created closure), which posts it to the destination processor instead of the original closure for `foo`.

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 k, char a[..])
{
    tp *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 (or portions of) data structure, then directly calls `foo`.

In order for things to work out correctly, the migration thread must be defined after both the original thread and the unpacking thread (or their prototypes) are defined, and it must have the same prototype 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.

4.7 Notes on Thread Argument Passing

Since closures may migrate, arguments to a spawned thread are normally passed by value, i.e., copyied into the closure. Migration handlers are generally needed when pointers are passed to spawned threads.

Cilk also allows threads to be called directly as C functions via the `call` and `tail_call` forms. An array argument to a direct or tail called thread must be passed by reference (see section 4.5.3). The Cilk preprocessor generates C code that is consistent with this interpretation of array arguments, i.e., if we have a thread declared as

```
thread foo (cont k, int a[10]) { ... }
```

and `b[10]` is an array, then `b[10]` will be copied into the closure 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 thread is spawned or called. When a structure is large, one may wish that the extra copying during direct or tail calls can be avoided. This can be achieved by specifying it as an array argument of 1 element. For example, instead of defining

```
thread foo (cont k, struct bar x) { ... }
```

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

```
thread foo (cont k, struct bar xp[1]) { ... }
```

4.8 Blocking Threads

4.9 Runtime System Interface

The Cilk runtime system implements mechanisms for thread scheduling, communication and synchronization. It also provides a library of runtime primitives for use by the programmer.

The Cilk runtime system has a completely independent interface, with naming conventions different from the Cilk language. However, most of the interface is hidden by Cilk. This section only describes the primitives visible to the programmer. For a complete reference to the runtime interface see chapter 5.

4.9.1 Entering the Scheduler

In order to execute threads, the following primitive must be invoked within the main control of a SPMD program:

`int RunScheduler (int` *mode*, `void` *tp* `(), int` *n*, arg_1, `...);`

Here *tp* is the first thread to run, *n* indicates how many arguments are supplied, and the remaining arguments are passed on as arguments to *tp*. By convention, the runtime system automatically supplies the first argument to *tp*, which must be a continuation, therefore *n* should be one less than the arity of *tp*. For example, if one invokes

```
result = RunScheduler (CILK_AUTO, foo, 2, 1, 10);
```

then there should be a corresponding thread defined as

```
thread foo (cont k, int a, int b) { ... }
```

The runtime system automatically defines a last thread which passes a continuation to `foo` as its first argument and awaits the result of the computation. Note that the argument sent to `k` will become the `result`, thus it must have type `int`. In addition, sending the result argument is also used as the completion signal of the multi-threaded execution, therefore it should not happen until all threads (except the last one) have been executed.

The *mode* argument should be one of

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

In the `CILK_AUTO` mode, the Cilk runtime system uses work stealing. In the `CILK_MANUAL` mode, work stealing is turned off. In either mode the heap will be reset when `RunScheduler` returns. If one wishes to keep the data structures allocated in the heap, which may be accessible either from the return value or from some global variables set during the execution, then `CILK_AUTO_KEEP` and `CILK_MANUAL_KEEP` should be specified instead.

4.9.2 Global Variables

The value of the following runtime system variable

```
extern int cilk_active_size;
```

must be set before entering the scheduler, which defines how many processors will participate in the computation. Its default value is 0, meaning all processors available. A positive integer tells the scheduler exactly how many processors to use.

The global variable

```
extern int Self;
```

is always set to the local processor's number.

4.9.3 Communication Primitives

The following primitives are available for sending arguments to different types of inlets:

```
void SendCharArgument (ContinuationT k, char value);
void SendShortArgument (ContinuationT k, short value);
void SendWordArgument (ContinuationT k, Word value);
void SendDoubleWordArgument (ContinuationT k, DoubleWord value);
void SendFloatArgument (ContinuationT k, float value);
void SendDoubleArgument (ContinuationT k, double value);
void SendArrayArgument (ContinuationT k, char *array, int size);
```

In the last form, the *size* of the array is in units of bytes.

Note that the Cilk preprocessor does not check which primitive is used for sending a thread argument, therefore it is the programmer's responsibility to always use the primitive that matches the type of the thread argument.

Note that within thread creation, for example

```
spawn foo (1, ?x[SIZE]);
```

it is only possible to capture a single continuation x pointing to the beginning of the array. However, sometimes it may be desirable to send an array argument element-wise or in pieces. To this purpose the primitive

`IndexContinuation (ContinuationT` *k,* `int` *i,* `int` *size*`);`

is provided. It returns a new continuation pointing to the *i*th element (counting from the reference point of *k*), where each element has *size* bytes.

Primitives for sending arguments to accumulator inlets are currently restricted to `Word`, `DoubleWord`, `float`, or `double` arguments:

`void AccumulateWord (ContinuationT` *k,* `AccumWordOp` *op,* `int` *value*`);`
`void AccumulateDoubleWord (ContinuationT` *k,* `AccumDoubleWordOp` *op,* `long long` *value*`);`
`void AccumulateFloat (ContinuationT` *k,* `AccumFloatOp` *op,* `float` *value*`);`
`void AccumulateDouble (ContinuationT` *k,* `AccumDoubleOp` *op,* `double` *value*`);`

The following accumulation operators are predefined:

`accum_word_add`, `accum_word_mul`,
`accum_word_and`, `accum_word_or`,
`accum_word_min`, `accum_word_max`,
`accum_doubleword_add`, `accum_doubleword_mul`,
`accum_doubleword_and`, `accum_doubleword_or`,
`accum_doubleword_min`, `accum_doubleword_max`,
`accum_float_add`, `accum_float_mul`,
`accum_float_min`, `accum_float_max`,
`accum_double_add`, `accum_double_mul`,
`accum_double_min`, `accum_double_max`

Additional accumulation operators can be defined by the user, see chapter 5 for details.

Finally, the primitive

`void Signal (ContinuationT` *k*`);`

can be used to send a signal.

4.9.4 Data Structure Primitives

Data primitives in Cilk are currently limited to local objects (useful together with migration threads) and global pointers.

The primitives

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

allocates or releases a block of memory, where *size* is in units of `Words`. The storage allocated are taken from the Cilk runtime heap, which will be reset after each run of the scheduler by default. Therefore if one wishes to carry them over to the next run of the scheduler, the current `RunScheduler` must be invoked with either `CILK_AUTO_KEEP` or `CILK_MANUAL_KEEP` mode.

An object of the type

```
globptr
```

is a global pointer which packs a local pointer together with a processor number. The processor number and the local pointer can be retrieved using the following macros

`PN(`*gptr*`)`
`OFFSET(`*gptr*`)`

Currently there is no primitive that fetches or stores data using a global pointer[4], therefore in order to access data from a global pointer *gptr* one needs to post a thread to processor `PN(`*gptr*`)`.

4.10 C Code Generated by the Preprocessor

This section briefly describes the structure of the C code generated from *cilkpp*, the thread preprocessor[5].

Cilkpp's output format consists of three sections, corresponding to the three sections in the source program:

C code

`/* ----- Begin Threads Section ----- */`

Closure structure definitions
Prototypes
Function definitions derived from threads

`/* ----- End Threads Section ----- */`

C code

We shall use the following simple example program to illustrate the C code generated from *cilkpp*:

```
local thread sum (cont k, int x, int y);

thread fib (cont k, int n)
{
    if (n<2) SendWordArgument (k, n);
    else
    {   cont x, y;
        spawn_next sum (k, ?x, ?y);
        spawn fib (x, n-1);
        call fib (y, n-2);
    }
}
```

[4]these did not make into the initial release of Cilk^{TM} 1.2, but is likely to be added very soon.

[5]When invoking *cilkpp* with the intention of reading the output, the **-N** switch should be used, which suppresses *cpp* line directives that would otherwise be generated by default. The UNIX program *indent* can be used to set the C code in a form that is easy to read.

4.10.1 Closure Structure Definitions

The structure definition of the closure for each thread is derived from its prototype. For example, the thread

```
local thread sum (cont k, int x, int y);
```

Generates the following type definition for its closure

```
typedef struct {
    ClosureT _header;
    ContinuationT k;
    int x;
    int y;
  } _sum_closure;
```

The first field of the structure contains additional information maintained by the runtime system, and is common to all closures. The thread argument variable names are used for the members of the structure. Note that new variable names generated by *cilkpp* are usually prefixed by _ to avoid potential conflicts with user defined variables.

Every thread argument has a member with the same type in the closure structure. The only exception is a signal argument, with is always left out. A variable-length argument such as `int a[]` has a member declaration in the form of `int a[0]`.

4.10.2 Prototypes

Cilkpp normally derives two functions from each thread. The first function is for general entry from the scheduler, which takes as its only argument the pointer to the closure. The second function is for direct entry from calls. The prototypes of all derived functions are declared before the function definitions. For example, the following are two prototypes are generated for thread `fib`:

```
extern void fib (_fib_closure *_cp);
extern void fib_fast (ContinuationT k, int n);
```

Note that the direct entry function is named `fib_fast` and takes all thread arguments.

In case a thread `foo` has a migration thread, an additional prototype is also declared for the function `foo_pack`, which is the general entry function derived from the migration thread:

```
ClosureT *foo_pack (_foo_closure *_cp);
```

Since the migration thread is only used to unpack the old closure, there is no direct entry function generated. Note that `foo_pack` is expected to return a pointer to a closure.

4.10.3 Function Definitions Derived from Threads

The general entry function first fetches all arguments from the closure, and deallocates the closure at the end (in fact, `FreeClosure` is inserted before every `return` statement). For example, below is the one for the `fib` thread:

```
void fib (_fib_closure * _cp)
{
    ContinuationT k = _cp->k;
    int n = _cp->n;

    ...

    FreeClosure ((ClosureT*) _cp);
}
```

The direct entry function differs from the general one only in that arguments are passed directly instead of fetched from the closure, and that no closure is deallocated since none is involved:

```
void fib_fast (ContinuationT k, int n)
{
    ...
}
```

An array argument is not fetched from the closure, instead a pointer is initialized to point to the array in the closure. For example, corresponding to the argument `int a[SIZE]`, within the general entry there will be a line:

```
int *a = _cp->a;
```

Array argument cannot be passed to direct entry functions.

Spawn

Spawning a thread is expanded into C code to allocate the closure, initialize the closure with supplied arguments, capturing continuations for the missing arguments, and posting the closure if possible. For example, the line

```
spawn_next sum (k, ?x, ?y);
```

becomes

```
_sum_cp_0 = (_sum_closure *)
    NewClosure ((ThreadT)sum, 0 + sizeof(int) + sizeof (int),
                sizeof (_sum_closure), 0);
_sum_cp_0->k = k;
x = MakeContinuation
        ((ClosureT *) _sum_cp_0, ARG_INDEX (_sum_cp_0, x));
y = MakeContinuation
        ((ClosureT *) _sum_cp_0, ARG_INDEX (_sum_cp_0, y));
SetClosureColor ((ClosureT *) _sum_cp_0, ClosureColorLocal);
```

after the *cilkpp* expansion. Note that

- The size of the closure is the number of bytes of the closure structure, plus the actual size of an additional variable-size array argument if any;

- The initial join count is calculated as the total bytes in all the missing arguments;
- The last argument to NewClosure (*childp*) is 0 if the closure is created via spawn_next, 1 if via spawn;
- ARG_INDEX(cp, x) is the byte offset of member x from the end of the common closure header;
- If a waiting closure is created, as in this example, the color of the closure is set according to supplied annotations: ClosureColorNormal is the default, ClosureColorLocal if the closure is declared local, etc. (see section 5.4 for definition of other colors).

If a closure is created with all the arguments, then the closure color will not be set, instead the appropriate version of PostClosure* will be called after the code generated from the trailing statement (if any).

Direct Call

A direct call is simply a call to the direct entry function. For example, the line

```
call fib (y, n-2);
```

Generates the following C statements:

```
TailCall ();
fib_fast (y, n-1);
```

Here the TailCall is issued to indicate calling a thread as the start of a new function invocation. If call_next is used then it will not be generated.

Tail Call

For efficiency, a tail call is implemented as a jump to the beginning of the thread after resetting the arguments. For example, If we replace the call statement in the fib thread to tail_call, then the following code will be generated (we only show the general entry function, the same happens with the direct entry function):

```
void fib(_fib_closure *_cp)
{
    /* fetching arguments from the closure */
    ...

    fib_direct_entry:
    {
        if (n<2) SendWordArgument (k, n);
        else
        {
            ...

            /* below are the lines generated for the tail_call */
            {
```

```
                ContinuationT k_tmp = y;
                int n_tmp = n - 1;

                k = k_tmp;
                n = n_tmp;
                TailCall ();
                goto fib_direct_entry;
            }
        }
    }
    FreeClosure ((ClosureT*) _cp);
}
```

Note that the label `fib_direct_entry` is inserted right after the code to fetch arguments from the closure, but `FreeClosure` is not part of the tail call loop. At the place where the tail call occurs, arguments are reassigned via temporaries to ensure correctness in case two arguments are swapped by the tail call.

4.11 Language Reference Manual Change Log

```
$Log: lanref.tex,v $
% Revision 1.18  1995/01/03  21:16:54  randall
% changed color constants.
%
% Revision 1.17  1994/11/06  02:19:33  randall
% Added AccumulateDoubleWord, SendDoubleArgument and AccumulateDouble to doc.
%
% Revision 1.16  1994/11/03  21:26:10  zhou
% Better example to illustrate the use of migration threads.
%
% Revision 1.15  1994/11/03  01:51:09  randall
% Updated everything to 1.2.  Added changes in 1.2 to the change log.
%
% Revision 1.14  1994/11/03  00:35:37  randall
% Added SendFloatArgument and AccumulateFloat to manual.
%
% Revision 1.13  1994/09/28  19:14:21  randall
% changed 1.0 -> 1.1
%
% Revision 1.12  1994/09/28  18:20:36  zhou
% Modify thread creation syntax, added a section on passing thread arguments
%
% Revision 1.11  1994/09/06  17:40:24  bradley
% Many small changes from my things-to-do list.
%
```

```
% Revision 1.10  1994/09/02  01:31:40  zhou
% Changed tpp to cilkpp everywhere
%
% Revision 1.9  1994/08/29  16:53:51  zhou
% fix some formatting problems
%
% Revision 1.8  1994/08/29  16:43:06  zhou
% add description of variables cilk_active_size and Self.
%
% Revision 1.7  1994/08/29  15:18:31  zhou
% Add label to chapter lanref
%
% Revision 1.6  1994/08/26  19:22:36  randall
% Added change log to the tutorial.
% Did some editing of the language reference.
%
% Revision 1.5  1994/08/26  15:22:25  zhou
% Some additions suggested by Keith
%
% Revision 1.4  1994/08/26  15:04:10  zhou
% Add section for data primitives
%
% Revision 1.3  1994/08/25  14:55:29  randall
% Made all of the undefined \ref{}s work.
%
% Revision 1.2  1994/08/24  15:23:15  bradley
% Added a changelog to langref.tex
% changed mac.tex to defs.tex.
%
```

Chapter 5

Specification of the Cilk Runtime System

5.1 Errata

Currently this document does not uniformly and accurately reflect the fact that we use the number of bytes received to implement the join-count.

5.2 Introduction

The Cilk$^{\text{TM}}$ 1.2 (Version $\beta 1$) run-time-system provides an execution environment for continuation-style threads written in the C programming language. The C code may be written directly by a human or it may be generated by a threaded-C preprocessor such as `cilkpp` (see Chapter 4.) This document specifies the run-time system. If you implement these functions then you should be able to run Cilk$^{\text{TM}}$ 1.2 programs. If you generate correct calls to these functions, then your program will run on any compliant Cilk$^{\text{TM}}$ 1.2 run-time-system.

The abstract programming model provided by Cilk$^{\text{TM}}$ 1.2 is a dataflowlike procedure call tree, where each procedure consists of a collection of threads. The entire computation consists of a directed acyclic graph (DAG) of *threads*. A thread is a piece of code, implemented as a C function. Each thread runs only after all of its DAG predecessors have run. Each DAG predecessor of a thread can communicate with it by sending it some arguments. The DAG does not exist all at once. A program can build the DAG on the fly. Furthermore, the DAG is organized into *levels* according the procedure call tree. The procedure call tree has little to do with the C functions used to implement the program. Instead, C functions are used to implement threads. A *procedure* is collection of threads with some dependencies among them. The procedures themselves are threaded together via procedure-calls.

Rather than keeping track of the degree of each node in the DAG, our run-time system keeps track of something else called the *join-counter*. The join-counter is a variable kept for each closure. As each predecessor of a closure executes, the join-counter is decremented. Since the join counter can be decremented by any positive integer, the join-counter does not specify the in-degree, but rather specifies the amount of decrementation that must occur before a thread runs.

The life cycle of a thread consists of

- Create the thread. When a thread is created, it is either created as a *child thread* (i.e, the thread is *spawned*) or a sibling thread.

- Modify the thread. After the thread is created and before it has been posted, it can be modified in various ways.

- Send values to the thread. As the predecessors of a thread execute, they send values to the thread, and decrement its join counter by appropriate values. Not every thread has values sent to it (in particular, a thread whose join-count is intially zero skips this step.)

- Post the thread. If the join-count of the thread is initially zero, then the thread is posted explicitly by the program. If the join-count of the thread is positive, then when enough values have been sent to the thread, the thread is posted automatically by the scheduler. A thread cannot be executed until it is posted. The closure can be moved to another processor only after it has been posted. A closure can be annotated ("*colored*") in various special ways, so that, for example, it will run on a specific processor.

- Execute the thread. After the thread has been posted, the thread can be run by the scheduler. The scheduler has some leeway about when it will actually run a thread, and it treats child threads differently from sibling threads (preferring to run a child before its parent, and to finish one child before starting another, although in the search for parallelism these rules may be relaxed.)

- Deallocate the storage for the thread. Note that is the responsibility of the thread code to deallocate its own closure before it returns. The runtime system does not automatically deallocate closures that have completed.

For certain threads, this lifecycle can be shortened. In particular for a thread that, as its last step, starts the execution of another thread (with in-degree of one), there is no need to invoke the scheduler. The C function can simply start executing the code for the next thread.

For certain programs we can guarantee space and time bounds, but we don't require the user to write such programs. Some of the requirements may be enforced by the scheduler. In particular, the scheduler is permitted to run a subtree without allowing other subtrees to make progress. Thus a non-strict program may deadlock. The current scheduler does not deadlock for such programs, but this specification allows us to write a scheduler that would deadlock. We expect to experiment with other schedulers in the future. In particular, we may eventually specify a scheduler that allows non-strict programs to run.

5.3 Data Types and Constants

Cilk™ 1.2 employs three data types, visible to the user, to represent these computations.

- A *closure* is the data structure used to represent a vertex in the DAG of the program. A closure is represented by a contiguous block of memory, starting with some header information maintained by the implementation, followed by space for the arguments. The header is of type `closureT`, which is a structure typedef. The structure contains an field named `args` which is an array of characters of length 0, the address of which is the beginning of the space to be used for the arguments. The `args` field of any closure is maximally aligned to avoid alignment errors (Thus on the SPARC, `args` is aligned to 64-bit word boundaries).

- A *continuation* is the data structure used to represent an incoming edge to a vertex of the DAG.

- A *thread* is the code which is run when a vertex of the DAG has been enabled.

Closures come in several *colors*, and have an extra integer associated with them, the *closure-info*. The colors are as follows:

`ClosureColorNormal`: A closure that can be stolen.[1]

`ClosureColorLocal`: A closure that must be executed on the same processor on which it was created.[2]

`ClosureColorPostNode`: A closure that must be executed on a particular named processor. The processor is specified by the closure-info.

`ClosureColorMigrationThread`: A migration closure is the same as a `ClosureColorNormal` closure, except that it uses the closure-info field to specify how to move the closure to another processor. The closure-info field contains a pointer (cast into an `int`) to a C function, called the *packing routine*. The packing routine takes the closure as its only argument, and returns another closure. The argument closure may contain, for example, pointers into the heap. The returned closure must be flat (i.e., all the required data must be represented *by value* in the block of memory allocated to the closure. The returned closure's thread typically points to an unpacking routine to be executed at the remote processor (e.g., to convert the flat representation back into a data structure on the heap.) The packing routine should use `FreeClosure()` to deallocate its argument. The unpacking routine typically posts its new closure, using the original thread, and should color the new closure `ClosureColorLocal` in order to guarantee that the closure will not be moved again before it is actually executed. Thus we have

```
ClosureT *packer(ClosureT *closure);
```

Rationale: Yuli observed that we might want migration threads that are posted on specific nodes. Bradley thinks that if the user wants to post a thread on a specific node, then the user should make the closure flat to start with, rather than writing a migration thread to pack up a structure. The packing is valuable because it is done only when something is moved. If you know that the work will move, then just do the packing immediately.

`ClosureColorHandlerThread`: A closure that should be executed as soon as possible on a particular processor. In particular, once all the predecessors have run, we prefer to execute a `ClosureColorHandlerThread` closure before executing any non-`ClosureColorHandlerThread` closures. The closure-info specifies on which processor the closure must be executed.

We use some type qualifiers borrowed from Strata:

GLOBAL means that the declared type has global or `extern` scope.

ATOMIC means that a function does not poll the networks.

INLINE means that there is an inlineable version of the procedure.

---

[1] The `ClosureColorNormal` color used to be called `STEAL`.
[2] The `ClosureColorLocal` color used to be called `NO_STEAL`.

5.4 Procedures

```
GLOBAL ATOMIC INLINE ClosureT *
  NewClosure ( ThreadT thread, int join_count, int size, int child_p );
```

effect: Create a new closure.

- The `join_count` specifies the initial join-count of the new closure. The value of `join_count` must be nonnegative. Note that even if `join_count` is zero, the closure is not posted into a ready queue.
- The `size` specifies the size of the closure block, measured in bytes. If you want to reserve space for n bytes of arguments, then call `NewClosure()` with the size set to `sizeof(ClosureT)` $+ n$. The value of `size` must be no less than `sizeof(ClosureT)`.
- The `child_p` specifies whether the new thread belongs to this level (`child_p==0`) or to the next deeper level in the call tree (`child_p==1`). The value of `child_p` must be zero or one.

The beginning of the block of memory used to store the `size` bytes of arguments is available as a field `args` in the returned closure.

note: After creating the closure, but before posting the closure, you may set the closure color, the closure info (for certain colors), and fill in some of the argument memory of the closure.

note: Typically the join-count is initialized to `size-sizeof(closureT)`, although sometimes the join-count is larger. In particular, when using accumulators and signals, the join-count should be increased according to number of accumulators and signals that are expected. See the accumulation and signalling functions described below.

```
GLOBAL ATOMIC INLINE void
  InitClosure ( ThreadT thread, int join_count, ClosureT *cp, int child_p );
```

effect: Create a closure by reusing a block of memory previously obtained from a call to `NewClosure()`. (Compare to `NewClosure()`, which creates a closure on a block of memory that the system newly allocates.) The `thread` specifies which thread will be run when the thread is posted and scheduled to execute. The `join_count` specifies the initial value of the join counter (just as for `NewClosure()`). The `child_p` specifies whether this is a sibling or spawned thread. The size of the closure is the same as the size of the original closure (and so, for example, do not try to modify the closure header to manipulate the size of the closure.)

`InitClosure()` allows the user to avoid the overhead of doing

```
FreeClosure(cp);
cp = NewClosure(...);
```

if both closures are the same size.

requires: The block must have been obtained by a previous call the `NewClosure()`.

rationale: The user could conceivably want to provide arbitrary memory for this closure. However, that would require reinitializing parts of the header that might not need reinitializaiton, and besides the system might want to put all the closures in a special area of memory, or index them by small integers.

```
GLOBAL ATOMIC INLINE void
  FreeClosure ( ClosureT *closure );
```

effect: Deallocates storage for this closure. All thread code should deallocate its closure before returning, unless it reuses the closure storage (with `InitClosure()`.)

requires: The closure must not be used in the future. This means, among other things, that the following must be true:

1. No memory loads or stores will, in the future, be made into the block of memory representing the closure (i.e., you should be finished using the closure fields);
2. No existing continuation pointing to this closure will be used in a SendArgument call in the future.
3. The closure is not currently posted anywhere. (That is, either the closure's join-count never went to zero, or else the thread associated with the closure has been run.)

```
GLOBAL ATOMIC INLINE void
  TailCall ( void );
```

effect: Notifies the run-time system that the C function has started executing another thread, and that the new thread is a child of the old thread. (There is no need to tell the scheduler that anything has happened if the new thread is a sibling of the old thread. The C function can simply start executing the new thread.)

```
GLOBAL ATOMIC INLINE void
  SetClosureColor ( ClosureT *closure, ClosureColorT color );
```

effect: Sets the color of the closure. The color of the closure determines how that closure will be posted when its join-count reaches zero. Setting the color to `ClosureColorX` tells the scheduler to post the closure with the `PostClosureX` routine (see below).

requires: The closure must not have been posted yet.

```
GLOBAL ATOMIC INLINE void
  SetClosureInfo ( ClosureT *closure, int info );
```

effect: Sets the closure-info of the closure. (Recall that closures of color `ClosureColorPostNode` and `ClosureColorHandlerThread` both use the closure-info to specify which processor the thread must be executed on. Closures of color `ClosureColorMigrationThread` use the closure-info to specify the packing routine. The closure-info is ignored for all other colors.)

requires: The closure must not have been posted yet.

```
GLOBAL ATOMIC INLINE void
  SetClosureThread ( ClosureT *closure, ThreadT *tp );
```

effect: Sets the thread (i.e., the C procedure) that will be run when the closure is posted.

requires: The closure must not have been posted yet.

```
GLOBAL ATOMIC INLINE void
  SetClosureJoin ( ClosureT *closure, int join );
```

effect: Sets the join counter of the closure. If you set the join counter to zero, you will need to manually post the closure.

requires: The closure must not have been posted yet, nor must the join counter have been manipulated by calls to a Send-Argument routine.

```
GLOBAL ATOMIC INLINE ContinuationT
  MakeContinuation ( ClosureT *closure, int index );
```

effect: Makes a continuation for the given argument index in the given closure. Index is an offset in bytes from the beginning of the argument list.

usage note: We suggest the following strategy for making it easy to name the arguments, handle alignment issues, and otherwise manage the block of memory that holds arguments. Cast the value returned from `NewClosure()` to a structure pointer, for example:

```
typedef struct {
  ClosureT c;
  int x;
  char y;
  double z;
} CT007;
...
  { CT007 cl = (CT007*)NewClosure(&th, 2, sizeof(CT007), 0);
    ContinuationT c1
        = MakeContinuation(&cl->c,
                           ((char*)&cl->x)-((char*)cl->args));
    ContinuationT c2
       = MakeContinuation(&cl->c,
                          ((char*)&cl->z)-((char*)cl->args));
    cl->y = 'b';
...
```

```
GLOBAL ATOMIC INLINE ContinuationT
  IndexContinuation ( ContinuationT k, Word index, Word size );
```

effect: Make a new continuation from an old continuation and an offset (where the offset is specified as index into an array of elements, each of size `size`). If `k` was created with offset `i` in a closure, then `IndexContinuation(k,j,s)` has offset $\mathtt{i} + \mathtt{j} \cdot \mathtt{s}$ in the same closure. That is,

`IndexContinuation(MakeContinuation(c,i),j,s)` $\equiv_{\text{def}}$ `MakeContinuation(c,i+j*s).`

```
GLOBAL ATOMIC INLINE void
  PostClosure ( ClosureT *closure );
```

effect: Posts the specified closure. If you know the color of the closure, then higher performance can be obtained by calling the appropriate specialized posting routine described below.

requires: The color (and the info, if used by that color), must be set.

note: The scheduler may use `PostClosure()` to post a closure when its join count goes to zero.

```
GLOBAL ATOMIC INLINE void
  PostClosureNormal ( ClosureT *closure );
```

effect: Sets the color of `closure` to be `ClosureColorNormal` and posts it.

```
GLOBAL ATOMIC INLINE void
  PostClosureLocal ( ClosureT *closure );
```

effect: Sets the color of `closure` to `ClosureColorLocal` and posts it.

```
GLOBAL ATOMIC INLINE void
  PostClosureNode ( ClosureT *closure, int pn );
```

effect: Posts a `ClosureColorPostNode` closure, and guarantees that the closure will executed only on the processor specified by `pn`. This is done by setting the closure's color to `ClosureColorPostNode` and the closure's info field to `pn`.

```
GLOBAL ATOMIC INLINE void
  PostClosureHandler ( ClosureT *closure, int pn );
```

effect: Sets the color of `closure` to `ClosureColorHandlerThread` and posts it so that it will run on processor `pn`. (This is done by setting the closure's info field to `pn`.)

```
GLOBAL ATOMIC INLINE void
  PostClosureMigration (ClosureT *cp, MigrationT mt );
```

effect: Sets the color of `closure` to `ClosureColorMigrationThread` and posts it. The closure's info field is set to be the `mt`.

issue: Need to define `MigrationT`.

```
GLOBAL INLINE void
  SendCharArgument ( ContinuationT continuation, char value );
```

effect: Send an 8-bit value to a continuation (decrementing the join counter of the closure, and posting the closure if the join-count becomes zero.)

```
GLOBAL INLINE void
  SendShortArgument ( ContinuationT continuation, short value );
```

effect: Send a 16-bit value to a continuation (decrementing the join counter of the closure by 2, and posting the closure if the join-count becomes zero.)

requires: The `continuation` must have been created with a 16-bit aligned `index`. (I.e., the `index` used for `MakeContinuation` must have been zero modulo 2.)

```
GLOBAL INLINE void
  SendWordArgument ( ContinuationT continuation, Word value );
```

effect: Send a 32-bit value to a continuation (decrementing the join counter of the closure by 4, and posting the closure if the join-count becomes zero.)

requires: The `continuation` must have been created with a 32-bit aligned `index`. (I.e., the `index` used for `MakeContinuation` must have been zero modulo 4.)

```
GLOBAL INLINE void
  SendDoubleWordArgument ( ContinuationT continuation, DoubleWord value );
```

effect: Send a 64-bit value to a continuation (decrementing the join counter of the closure by 8, and posting the closure if the join-count becomes zero.)

requires: The `continuation` must have been created with a 64-bit aligned `index`. (I.e., the `index` used for `MakeContinuation` must have been zero modulo 8.)

```
GLOBAL INLINE void
  SendFloatArgument ( ContinuationT continuation, float value );
```

effect: Send a 32-bit floating-point value to a continuation (decrementing the join counter of the closure by 4, and posting the closure if the join-count becomes zero.)

requires: The `continuation` must have been created with a 32-bit aligned `index`. (I.e., the `index` used for `MakeContinuation` must have been zero modulo 4.)

```
GLOBAL INLINE void
  SendDoubleArgument ( ContinuationT continuation, double value );
```

effect: Send a 64-bit floating-point value to a continuation (decrementing the join counter of the closure by 8, and posting the closure if the join-count becomes zero.)

requires: The `continuation` must have been created with a 64-bit aligned `index`. (I.e., the `index` used for `MakeContinuation` must have been zero modulo 8.)

```
GLOBAL INLINE void
  SendArrayArgument ( ContinuationT continuation, char *array, int length );
```

effect: Send a block of memory to a continuation (decrementing the join counter of the closure by `length`, and posting the closure if the join-count becomes zero.) If the `length` is zero, the pointer `array` is not dereferenced. The `SendArrayArgument()` function can be used to send both structures and arrays.

performance: This routine is most efficient when both the `array` and the `index` used for `MakeContinuation` are aligned similarly with respect to 32-bit word boundaries (I.e., they are congruent modulo 4).

requires: The value of `length` must not be larger than the remaining join count.

```
GLOBAL INLINE void
  Signal (ContinuationT k);
```

effect: This routine sends a *signal* to a closure. A signal is a just like any other Send-Argument routine, except no data is sent. The join-count is decremented (by 4?), and if it becomes zero, the closure is posted.

issue: Should the join count decremented by 4 or by 1?

```
GLOBAL INLINE void
  AccumulateWord (ContinuationT k, AccumWordOp op, int delta);
```

effect: The closure memory element specified by the continuation `k` is modified according to `op` and `delta`. That is, if `a` points to the address referenced by the continuation `k`, then this routine calls

```
(*op)(delta,a);
```

The closure's join count is decremented (by 4?), and if it becomes zero, the closure is posted.

The Cilk system provides the following predefined functions.

`accum_word_add`**:** (addition)

`accum_word_mul`**:** (multiplication)

`accum_word_and`**:** (bitwise logical 'and')

`accum_word_or`**:** (bitwise logical 'or')

`accum_word_max`**:** (signed maximum)

`accum_word_min`**:** (signed minimum)

For example, `accum_word_add` could have been defined as

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

The user can also define and use his or her own accumulation operations.

requires: The continuation `k` must have been created with an index that is sufficiently aligned that the function `op` can access it without causing trouble. The system-defined operations require 32-bit alignment.

note: AccumWordOp is defined as:

```
typedef void (*AccumWordOp)(int, int*);
```

issue: Should the join count decremented by 4 or by 1?

```
GLOBAL INLINE void
  AccumulateDoubleWord (ContinuationT k, AccumDoubleWordOp op, long long delta);
```

effect: The closure memory element specified by the continuation `k` is modified according to `op` and `delta`. That is, if `a` points to the address referenced by the continuation `k`, then this routine calls

```
(*op)(delta,a);
```

The closure's join count is decremented (by 8), and if it becomes zero, the closure is posted.

The Cilk system provides the following predefined functions.

`accum_doubleword_add`**:** (addition)

`accum_doubleword_mul`**:** (multiplication)

`accum_doubleword_and`**:** (bitwise logical 'and')

`accum_doubleword_or`**:** (bitwise logical 'or')

`accum_doubleword_max`**:** (signed maximum)

`accum_doubleword_min`**:** (signed minimum)

requires: The continuation `k` must have been created with an index that is sufficiently aligned that the function `op` can access it without causing trouble. The system-defined operations require 64-bit alignment.

note: AccumDoubleWordOp is defined as:

```
typedef void (*AccumDoubleWordOp)(long long, long long*);
```

```
GLOBAL INLINE void
  AccumulateFloat (ContinuationT k, AccumFloatOp op, float delta);
```

effect: The closure memory element specified by the continuation `k` is modified according to `op` and `delta`. That is, if `a` points to the address referenced by the continuation `k`, then this routine calls

```
(*op)(delta,a);
```

The closure's join count is decremented (by 4), and if it becomes zero, the closure is posted.

The Cilk system provides the following predefined functions.

`accum_float_add`**:** (addition)

`accum_float_mul`**:** (multiplication)

`accum_float_max`**:** (maximum)

`accum_float_min`**:** (minimum)

requires: The continuation `k` must have been created with an index that is sufficiently aligned that the function `op` can access it without causing trouble. The system-defined operations require 32-bit alignment.

note: AccumFloatOp is defined as:

```
typedef void (*AccumFloatOp)(float, float*);
```

```
GLOBAL INLINE void
  AccumulateDouble (ContinuationT k, AccumDoubleOp op, double delta);
```

effect: The closure memory element specified by the continuation `k` is modified according to `op` and `delta`. That is, if `a` points to the address referenced by the continuation `k`, then this routine calls

```
(*op)(delta,a);
```

The closure's join count is decremented (by 8), and if it becomes zero, the closure is posted.

The Cilk system provides the following predefined functions.

`accum_double_add`**:** (addition)

`accum_double_mul`**:** (multiplication)

`accum_double_max`**:** (maximum)

`accum_double_min`**:** (minimum)

requires: The continuation `k` must have been created with an index that is sufficiently aligned that the function `op` can access it without causing trouble. The system-defined operations require 64-bit alignment.

note: AccumDoubleOp is defined as:

```
typedef void (*AccumDoubleOp)(double, double*);
```

```
GLOBAL int
  RunScheduler ( int mode, ThreadT root_thread, int num_args, ... );
```

effect: Starts up the scheduler and runs the thread `root_thread` with `num_args` words of arguments. Note that `num_args` (and the argument list) does NOT include the continuation which must be the first argument of `root_thread`. This continuation will be provided by RunScheduler. Note that this is a data-parallel operation so all processors involved must call it. (All arguments except `mode` are ignored on processors where `Self` $\neq 0$). Work-stealing is enabled if mode is `CILK_AUTO` or `CILK_AUTO_KEEP`, and the heap is kept around upon return if mode is `CILK_AUTO_KEEP` or `CILK_MANUAL_KEEP`.

requires: The value of `mode` must belong to the enumeration type

```
enum run_mode { CILK_AUTO, CILK_MANUAL, CILK_AUTO_KEEP, CILK_MANUAL_KEEP };
```

All processors must call `RunScheduler()` at about the same time.

The value of `cilk_active_size` must be valid.

issue: I need to define `Self`.

issue: Bradley believes that the specification of this routine needs to be rewritten. For example, what do we mean by "about the same time"?

```
GLOBAL void
  CilkInit( void );
```

meaning: Initializes the communication library that Cilk uses.

```
GLOBAL void
  CilkExit( int exitcode );
```

meaning: Ends the Cilk program. If `Exitcode` is zero, this is a normal exit and the processor that calls it will wait for the others to finish. If `exitcode` is non-zero, it is an error exit. (all processors will exit if any processor calls `CilkExit` with a non-zero argument.) This should be called after `RunScheduler` returns.

```
  extern int cilk_active_size;
```

meaning: The `cilk_active_size` specifies how many processors the user actually wants to use. The user can set this variable before calling `RunScheduler()`. The value must be a nonnegative value no greater than the number of actual processors. A zero value is used to specify as many processors as are actually available. A positive number specifies a particular number of processors to use.

The user may want to specify just a few processors in order to obtain data about the performance of his or her application as the number of processors is varied.

default: This variable defaults to zero.

```
extern void (*cilk_user_function)();
extern int n_threads_per_user_function;
```

meaning: The function `cilk_user_function` will be called periodically on each node by the scheduler so that the user may perform some periodic tasks. This function will be called more often if the value of `n_threads_per_user_function` is smaller. Both of these global variables must be set before the call to `RunScheduler`. They may have different values on different processors. Use `NULL` for the cilk_user_function in order to disable this feature.

default: `cilk_user_function` defaults to `NULL` and `n_threads_per_user_function` defaults to 20.

requires: These variables must not be modified by the user while the scheduler is running.

```
extern int migration_dest_node;
```

meaning: This variable holds the processor number of the destination processor for a migration thread. It is only valid inside migration handlers.

issue: This is used in the chess code for aborts. Is it useful as a general mechanism? Should we expose it?

5.5 Runtime Specification Change Log

```
$Log: runtime-spec.tex,v $
% Revision 1.12  1995/01/03  21:17:01  randall
% changed color constants.
%
% Revision 1.11  1994/11/06  02:19:43  randall
% Added AccumulateDoubleWord, SendDoubleArgument and AccumulateDouble to doc.
%
% Revision 1.10  1994/11/03  00:35:46  randall
% Added SendFloatArgument and AccumulateFloat to manual.
%
% Revision 1.9  1994/09/06  17:40:28  bradley
% Many small changes from my things-to-do list.
%
% Revision 1.8  1994/09/02  19:22:31  randall
% StrataInit -> CilkInit, same for exit.
%
% Revision 1.7  1994/09/02  01:31:43  zhou
% Changed tpp to cilkpp everywhere
%
% Revision 1.6  1994/08/26  19:16:30  bradley
% Most of the join-counter specification now talks about counting bytes.
%
% Revision 1.5  1994/08/26  18:01:46  bradley
% Add change log sections.
%
```

Bibliography

[BB94] Eric A. Brewer and Robert D. Blumofe. Strata: A multi-layer communications library. Technical report, MIT Laboratory for Computer Science, January 1994. To appear. Available from `ftp.lcs.mit.edu` via anonymous ftp, in directory `/pub/supertech/strata`.

[BL93] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing (STOC '93)*, pages 362–371, San Diego, California, May 1993.

[Blu92] Robert D. Blumofe. Managing storage for multithreaded computations. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1992. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-552.

[BP94] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing (HPDC '94)*, pages 96–105, San Francisco, California, August 1994.

[HZJ94a] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming based on continuation-passing threads. Computation Structures Group Memo 355, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1994. A shorter version will appear in Proc. of 2nd Int. Workshop on Massive Parallelism: Hardware, Software and Applications. Capri, Italy, Oct. 1994.

[HZJ94b] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming based on continuation-passing threads. In *Proc. of 2nd Int. Workshop on Massive Parallelism: Hardware, Software and Applications.*, Capri, Italy, October 1994. To appear.

[Kus94] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1994.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Symposium on Parallel Algorithms and Architectures (SPAA '92)*, pages 272–285, San Diego, California, June 1992.

[PJGT] Vijay S. Pande, Chris Joerg, Alexander Yu. Grosberg, and Toyoichi Tanaka. Enumeration of the hamiltonian walks on a cubic sublattice. To appear in Journal of Physcs A.

Appendix A

Cilk Implementor's Guide

Cilk™ 1.2 is the first Cilk release written to facilitate porting to different kinds of sequential and parallel machines. The code is accepted by any standard ANSI C compiler (we recommend `gcc`). In order to port Cilk easily, there are four steps you must accomplish; each step is an incremental change over the previous ones. The first step is to make Cilk run on one processor; the second one is to build a work-stealing version of Cilk; in the third step `PostClosureNode` is implemented, and finally you'll add time measurements in the last step. The porting process is mainly a *clever definition of macros*, as you shall see.

A.1 First step: get a sequential version of Cilk running

Your first task is to get Cilk running on your architecture without any of the work stealing and parallel code. This is very simple, since Cilk is written in ANSI C and requires no particular support from the underlying operating system or compiler: however some `gcc` extensions are supported in order to have better performances. First, decide a name for your architecture (e.g. `CM5`, or `PARAGON`) and edit the file `rts/conf.h`. This file contains some options you are expected to define in this first step.

Create a block of definitions just before the section marked `Statistics & Debugging` similar to the prototype included in the file. There are some options you might want in order to get a different behavior or better performance:

`IS_PARALLEL` Leave this option off for the moment. Enabling this option causes work stealing code to be compiled, and this requires your intervention. Wait until step 2 to turn this on.

`HAVE_LONG_LONG` Turn this on if your compiler supports very long integers (64 bits); also define `LONG_LONG` to be the C type of these integers (probably `long long`). `gcc` supports this type.

`CILK_PRINTF` This is a `printf`-like function Cilk uses for its diagnostic messages, statistics and such. `#define CILK_PRINTF printf` is probably good if your `printf` doesn't clobber the network with messages. Otherwise you should write your own function.

`CONTINUATION_HACK` Turn this option on if you have the `LONG_LONG` type and want to pack continuations to fit into this type. Otherwise, leave this option off. This option will allow the compiler to register-allocate continuations.

HAVE_INLINE Turn this option on if your compiler supports `inline` functions and you want to use them. There is a speedup for using this option, at expense of increased code size.

CILK_FAST_RNG Cilk uses an internal random number generator when `IS_PARALLEL==1`. There are two versions of this generator: a fast version and a portable one. The former assumes you have 64 bits `unsigned long long`, while the latter is more portable (but still assumes 32 bit `unsigned long`).

CILK_POST_NODE Leave this off until step 3.

CILK_ARCH_H Define this to be the name of the architecture-specific header file.

There are some more options you can define. It's advisable to have `CILK_SELFTEST==1` (this enables lots of consistency checks, valuable for debugging), and `CILK_TRACE` equal to some small integer (1 or 2); incrementing it gives more and more diagnostic messages, providing clues for debugging. Leave the timing-related options disabled for now.

Now you must create two files, `rts/arch-your_arch.c` and `rts/arch-your_arch.h`. In the `.h` file put something like

```
#define CILK_LONG_ALIGNMENT 4
#define CILK_SHORT_ALIGNMENT 2
#define CILK_LONG_LONG_ALIGNMENT 8
#define CILK_FLOAT_ALIGNMENT 4
#define CILK_DOUBLE_ALIGNMENT 8
```

The goal of these macros is to provide means of allocating variables without violating architectural requirements for alignment of variables. The definitions given should be good for most processors in use today. If you haven't got `LONG_LONG`, you don't need to define the corresponding alignment.

Also insert something like

```
#define CRITICAL_SECTION_BEGIN()
#define CRITICAL_SECTION_END()
```

These macros will not be used until later stages, but should be defined now in order to avoid compilation errors. You can start by making a copy of `rts/arch-sunos4.h` to have all these definitions in the right place.

Take the time to define the macro `Cilk_FlushStdout()`: this should make sure that all diagnostic messages are actually output, and internal buffers of `CILK_PRINTF` are emptied. Probably

```
#define Cilk_FlushStdout() fflush(stdout)
```

would suffice.

Then you must edit `rts/arch-your_arch.c` to be something like

```
#include <cilk.h>

int Self;

void Cilk_ArchSpecificInit()
```

```
{
    Cilk_PartitionSize = /* put here a function to retrieve the
                          * number of processors Cilk is running on */
    Self = /* put here a function to retrieve the processor number of
              * the local processor. */
}

void Cilk_ArchSpecificExit(int status)
{
    /* put here architecture-dependent exit code */
    exit(status);
}
```

Again, `rts/arch-sunos4.c` is a good example to start with. If your machine has only one processor, just let `Cilk_PartitionSize=1` and `Self=0`.

Now edit the `Makefile` to provide suitable flags to the C compiler. These flags should be added as `YOUR_ARCHITECTUREFLAGS` after the `UNIXFLAGS` definition. You should be sure to define your architecture's flag to be 1 (i.e., include the flag `-DYOUR_ARCHITECTURE=1` in your flag list). Also, you must add libraries for your machine to the `libraries` list. A good way to do this is to replace `cm5` with your architecture in the `libraries` rule and in the following three `libCilk` rules. This replacement will allow you to build libraries for your architecture as well as a standard Unix architecture. You must also change some of the rules in the `building executables` and `building binaries` to reflect your architecture's linking and compilation rules. Be sure to use `YOUR_ARCHITECTUREFLAGS` in the `building binaries` section. Finally, you should also make the same changes to the `Makefile` in the examples directory.

Now compile everything and go to the `examples` directory. You should be able to compile and run all files. Try `fib 0 25`, whose expected result is 75025, and `queens 0 8` (92 is the correct result). If you have `gcc` you should be able to compile and run `rts/testall.p`.

A.2 Second step: get a work-stealing scheduler running

If you machine is sequential, go to step 4. Otherwise enjoy: unlike the first stage, this stage requires some ingenuity and creativity . You will need to write code yourself, rather than simply cutting and pasting existing code. You'll demonstrate your ability to write compact, elegant and efficient code for the Cilk Runtime System!

First of all, turn on the option `IS_PARALLEL` in `conf.h`. Having done that, everything that can go wrong will go wrong, so you'd better increase the tracing level a bit.

Implement Remote Procedure Calls. Cilk is built over an abstraction of "Remote Procedure Calls". If the Cilk's Runtime System needs to communicate something to another processor, it invokes a C function (called a *handler* in Cilk's terminology) with the appropriate arguments. You'll need to figure out how to accomplish this effect on your system. We will provide some examples later; in the meantime please be patient until you learn more about handlers used by Cilk.

Cilk requires you to specify how to implement four types of handlers:

```
DECLARE_SENDARG_HANDLER(Cilk_SendWordArgumentHandler, long)
{
     SENDARG_HANDLER_BEGIN(long);
     *((long *) ((char *) cp + index)) = value;

     cp->join -= sizeof(long);
     assert(cp->join >= 0);

     if (!cp->join)
          PostClosure_nomsg(cp);
     SENDARG_HANDLER_END();
}
```

Figure A.1: An example of handler from the Cilk code

`SENDARG` **handlers** : used to communicate arguments to a remote closure.

`ACCUM` **handlers** : used to accumulate arguments in a remote closure. The various associative operations you can use to accumulate have been described earlier in this manual.

`SIGNAL` **handlers** : used to implement the `Signal` operation. These are degenerate `SENDARG` handlers, because they don't carry any value. Currently, there is only one handler of this type.

`GENERIC` **handlers** : used for various purposes. These handlers are functions of one `long` argument.

Moreover, the first two types have also an extended variant, since we imagine that very long C types, in our case `double` and `long long`, may require different treatment on some machines.

So, how do handlers work? Let's look at a specific example from the code, as in Figure A.1. A handler of type `SENDARG` needs three arguments: a `value` whose type is specified by the user, a variable `cp` of type `ClosureT *` and an integer `index`. Moreover the handler is invoked by the macro expansion

```
INVOKE_SENDARG_HANDLER(pn, Cilk_SendWordArgumentHandler, long,
                       value, cp, index);
```

Your goal is to implement four macros for each type of handler, plus one macro or function `Cilk_POLL()`, which cooperate to make things work. The basic idea is that macros of type `DECLARE` provide the function prototype of the handler; `BEGIN` macros declare additional arguments and read them from the network; `END` macros reverse any action done by the `BEGIN` macros; and finally `INVOKE` takes care of executing the handler on a remote processor passing the appropriate parameters. `Cilk_POLL` has a special meaning: if RPCs are implemented without using interrupts, this function must check for incoming requests and satisfy them. Otherwise if an incoming request suspends the program, `Cilk_POLL` is a no-operation. As you may guess, `Cilk_POLL` is called at appropriate points of the Cilk Runtime System.

For clarity, let's now look at some specific examples. Let's suppose your machine has *synchronous message passing*, i.e. you have three primitives *send*, *receive* and *test* to send, receive and test the

Handler's type	Arguments	Remarks
SENDARG	type value; ClosureT *cp; int index;	type is an argument to DECLARE and INVOKE macros.
SENDARG_X	type value; ClosureT *cp; int index;	type is double or LONG_LONG (if defined)
ACCUM	deltatype delta; optype op; ClosureT *cp; int index;	deltatype and optype are arguments to DECLARE and INVOKE macros.
ACCUM_X	deltatype delta; optype op; ClosureT *cp; int index;	deltatype is double or LONG_LONG (if defined)
SIGNAL	ClosureT *cp;	
GENERIC	int arg;	arg is a parameter of the DECLARE and BEGIN macros

Table A.1: Arguments to remote handlers, by handler's type

presence of messages. An example of implementation for this case is shown in Figure A.2. We use a protocol where a handler invocation packs the handler arguments in a suitable C structure, where the first field is common to all such structures and is a pointer to the handler. Note that since this implementation can send any C type, the extended handlers can be defined in terms of the basic ones.

An interesting variation on this theme is the Intel Paragon, which has the concept of a *message type*. This is a 32-bit field which can be used to keep the pointer to the handler.

Now suppose that you have *asynchronous message passing*, i.e. when a message arrives it interrupts the program and an appropriate user-defined function is called. The previous scheme still is a valid implementation, except for the following:

- `Cilk_POLL()` is now void, and the old `Cilk_POLL` becomes the user-defined function called at interrupt time.

- You should define the macros `CRITICAL_SECTION_BEGIN()` and `CRITICAL_SECTION_END()` to respectively disable reception of further messages and restore the state preceding the last `CRITICAL_SECTION_BEGIN()`. *Warning: the exact placement of these routines has not been tested because we have currently tested Cilk only on polling architectures.*

If your machine has remote procedure calls, you can use them. The existing implementation for the CM5 uses RPCs, so you might want to take a look at the existing code.

To conclude the section about handlers, Tables A.1 and A.2 summarize all handler-related macros you should implement.

```
/* this must be added to arch-your_arch.c */

static long buf[10]; /* maximum length of a message. This buffer should be
                     * larger if you want to use it also for work
                     * stealing, see below.
                     */

void Cilk_POLL(void)
{
     while (test()) {
          /* receive the message into buf */
          receive(buf);

          /* can't understand what's going on? This is equivalent
           * to:
           *
           * void (*fp)(long *) = (appropriate cast) buf[0];
           * (*fp)(buf);
           */
          (*(void (*)(long *))buf[0])(buf);
     }
} /* that's all the dispatcher */

/* the rest is part of arch-your_arch.h */
extern void Cilk_POLL(void);

#define DECLARE_SENDARG_HANDLER(name, type) \
     void name(long *lp)

#define SENDARG_HANDLER_BEGIN(type) \
     struct { long dummy; type value; ClosureT *cp; int index; } *sp = lp; \
     type value = sp->value;      \
     ClosureT *cp = sp->cp;       \
     int index = sp->index;

#define SENDARG_HANDLER_END()

#define INVOKE_SENDARG_HANDLER(pn, whichhandler, type, value, cp, index) \
{
     struct { void (*fp)(long *); type a; ClosureT *b; int c; } _s; \
     _s.fp = whichhandler; \
     _s.a = value;        \
     _s.b = cp;           \
     _s.c = index;        \
     send(pn, &_s, sizeof(_s)); \
}
```

Figure A.2: An example of SENDARG macros for a synchronous-messages architecture

```
#define DECLARE_SENDARG_HANDLER(name, type)
#define INVOKE_SENDARG_HANDLER(pn, whichhandler, type, value, cp, index)
#define SENDARG_HANDLER_BEGIN(type)
#define SENDARG_HANDLER_END()

#define DECLARE_SENDARG_XHANDLER(name, type)
#define INVOKE_SENDARG_XHANDLER(pn, whichhandler, type, value, cp, index)
#define SENDARG_XHANDLER_BEGIN(type)
#define SENDARG_XHANDLER_END()

#define DECLARE_ACCUM_HANDLER(name, deltatype, optype)
#define INVOKE_ACCUM_HANDLER(pn, whichhandler, op, type, delta, cp, index)
#define ACCUM_HANDLER_BEGIN(deltatype, optype)
#define ACCUM_HANDLER_END()

#define DECLARE_ACCUM_XHANDLER(name, deltatype, optype)
#define INVOKE_ACCUM_XHANDLER(pn, whichhandler, op, type, delta, cp, index)
#define ACCUM_XHANDLER_BEGIN(deltatype, optype)
#define ACCUM_XHANDLER_END()

#define DECLARE_SIGNAL_HANDLER(name)
#define INVOKE_SIGNAL_HANDLER(pn, whichhandler, cp)
#define SIGNAL_HANDLER_BEGIN()
#define SIGNAL_HANDLER_END()

#define GENERIC_HANDLER(name, arg)
#define GENERIC_HANDLER_BEGIN()
#define GENERIC_HANDLER_END()
#define INVOKE_GENERIC_HANDLER(pn, whichhandler, arg)
```

Table A.2: List of handler-related macros

Now provide barriers. In Cilk's jargon, a *barrier* is a synchronization point: the goal of a barrier is to make sure that a processor can cross the barrier only when *all* processors have reached it. You must define a function (or macro) `void Cilk_Barrier(void)` to do exactly this job: this function is probably already implemented in one of you system libraries. No special properties are required from a barrier: it doesn't need to poll for incoming messages or allow interrupts to occur (it can do these things if it wishes). We'd like to hear from you if you find any difficulty in implementing barriers (i.e. if your system doesn't already provide such functionality): we can design a general mechanism for making barriers using RPCs if they are not easily implementable.

We recognize that writing all the required macros is rather boring and error-prone. There is a special file `rts/test-handlers.c` which tests all handlers and barriers. It's wise to run it at this point and look at the output. If anything goes wrong it's better to increase the tracing level to 1 and look at the more detailed output.

Now implement work-stealing. You need to implement the work-stealing protocol, which is very simple: when there is no ready closure on some processor (the *thief*), the scheduler calls `steal()`, which in turn sets the global variable `steal_request_pending` to 1, and invokes the handler `steal_request_handler` on some random processor (the *victim*). `steal_request_handler` calls `get_stealable_closure()` which returns `cp`, a pointer to a closure. There are now two cases: if `cp` is `NULL`, there is no stealable closure, and the handler `steal_failure_handler` must be invoked in the thief processor. Otherwise the closure pointed to by `cp` is to be sent to the thief, invoking `steal_success_handler`.

The Figures A.3 and A.4 show a sketch of an implementation; for your convenience the same code is in the file `rts/cilk.c`, which you should modify at this point. Some random comments to the proposed code:

- `HANDLER` is not a macro or keyword. It's simply a clue that you must construct a handler accepting some arguments in the same way you that defined the macros above.
- Cilk does not require (but allows) the complete closure to be sent over the network : the first `CLOSURE_NONXMITTED_WORDS` words aren't transmitted, and the first transmitted field is given by the macro `CLOSURE_FIRST_XMITTED_FIELD`.
- If you build a single message containing the closure, make sure you allocate a large buffer on the receiving side (`MAX_CLOSURE_SIZE` longs is large enough). In this case you'll probably gain in performance by sending the whole closure instead of copying parts of it into a buffer and sending the parts separately.

Now try the same examples of step 1 and observe the speedup. Remark: `tree.p` isn't going to work until the next step is complete.

A.3 Step three: implement `PostClosureNode`

In this step you are going to implement the protocol which allows the user to specify which processor a given thread is to be run on. There are actually two different protocols, depending on whether the thread to be remotely posted is a ordinary thread or an high-priority one (called *handler thread* in the

```
static void HANDLER(steal_failure_handler)
{
     steal_request_pending = 0;
}

static void HANDLER(steal_success_handler, int size)
{
     ClosureT *cp, *incoming_cp;

     cp = (ClosureT *) alloc_block(size + CLOSURE_NONXMITTED_WORDS);

     /*
      * put here code to let incoming_cp point to the incoming closure;
      * maybe let incoming_cp point to a static buffer and read
      * the data from the network into that buffer, you figure it out.
      */

     /* copy the incoming closure into Cilk buffers */
     memcpy((char *) &CLOSURE_FIRST_XMITTED_FIELD(*cp),
            (char *) &CLOSURE_FIRST_XMITTED_FIELD(*(ClosureT *) incoming_cp),
            size * sizeof(long));

     /* some bookkeeping, don't forget it */
     steal_request_pending = 0;
     Cilk_AdoptClosure(cp);
     PostClosureLocal(cp);
     WHEN_CILK_STATS(Cilk_num_migrated_threads++);
}
```

Figure A.3: Skeleton of work-stealing code, part 1/2

```
static void HANDLER(steal_request_handler, int pn)
{
     ClosureT *cp = get_stealable_closure(pn);
     int size;

     if (cp == NULL) {
          /* put code here to invoke steal_failure_handler on pn */
     } else {
          size = cp->size - CLOSURE_NONXMITTED_WORDS;
          /*
           * put code here to invoke steal_success_handler on pn,
           * in such a way that it can access the variable `size' and
           * read 'size' longs starting from the address
           * CLOSURE_FIRST_XMITTED_FIELD(*cp)
           */
          /* some bookkeeping, don't forget it */
          FreeClosure(cp);
     }
}

static void steal(void)
{
     int victim;

     if (steal_request_pending)
          return;

     /* Chose a random target other than Self */
     victim = Cilk_Random() % cilk_active_size;
     if (victim == Self)
          return;

     steal_request_pending = 1;

     /*
      * put here code to invoke steal_request_handler on victim,
      * passing Self as argument
      */
}
```

Figure A.4: Skeleton of work-stealing code, part 2/2 current release[1]). The two protocols are very similar, since only the final action is different; hence we'll describe only the protocol for ordinary threads, remarking the differences when needed.

The protocol begins with the user's program invoking the function `PostClosureNode()` (resp. `PostClosureHandler()` for the second protocol). This function in turn builds an appropriate packet containing the closure and sends it to destination processor, invoking `post_rqsthndlr` (resp. `posthandler_rqsthndlr`). `post_rqsthndlr` adopts the closure and posts it locally (resp. enqueues it in the high-priority queue).

As you can see, the protocol is very simple, but there are machines (such as the CM5) on which it wouldn't work, or wouldn't be very efficient, and a more elaborate protocol is needed. The problem is that the source processor is sending to the destination a large, unexpected packet: this packet could fill buffers or deadlock the network, resulting in an error. Even if this protocol works, it might be more efficient, if the architecture supports it, to deliver the closure directly to its final destination, instead of filling up some intermediate buffer and copying it. Therefore we'll describe the protocol used by the CM5, in case you need to implement something similar on your architecture.

Protocol for the CM5. `PostClosureNode()` invokes `post_rqsthndlr` on the destination processor, without sending it the actual closure, but communicating I) a pointer to it, `cp` (note that this pointer is meaningless on the destination processor); II) the size of the closure; III) the source processor identifier. `post_rqsthndlr` (on the destination processor) allocates space to hold the incoming closure, and tells the communication library to store incoming data in that space. Then it invokes `post_reply_handler` on the source processor, giving it back `cp`. Now, contrary to what you might expect, `post_reply_handler` *does not* send the closure pointed to by `cp` immediately to the destination processor. Rather, it posts a thread on the local high-priority queue. This thread (`post_thread`) will be run later by the scheduler and will send the closure to destination. At the end of the transmission, `post_final_handler` is invoked on the destination processor to adopt the closure and post it. The reason of the delayed transmission of the closure is very CM5-specific: since the CM5 does not guarantee that the C stack can grow enough to allow all messages to be received from the network. On your machine you might want to send the closure immediately.

Figures A.5 and A.6 contain a skeleton of the code you must write to implement the simpler protocol.

`testall` is the right program to run now, if you have `LONG_LONG` (sorry for this, a more portable `testall.p` will be released in the next version). And again, all examples should run fine: try `tree 0 14`, and expect $2^14 = 16384$ leaves.

A.4 Step four: add time statistics

This step isn't really needed for full functionality of Cilk; however having these kinds of statistics is a good thing anyway, and you should dedicate some time to make them work, once and forever.

Cilk supports two kind of time measurements: a coarse-grained timing of the total execution and a fine-grained measurement of the execution time of each thread, which is used to determine the critical path and the work done by every processor.

[1]This name is unfortunate, because an high-level user-visible feature has the same name as a low-level implementing detail. We'll probably remove the naming conflict in a future release.

```
void PostClosureNode(ClosureT *cp, int pn)
{
     assert(pn >= 0);
     assert(pn < cilk_active_size);

     CRITICAL_SECTION_BEGIN();
     if (pn == Self)
          PostClosureLocal(cp);
     else {
          cp->info = pn;
          /*
           * put here the code to send the closure to pn
           * and invoke post_rqsthdlr(cp->size). Don't forget to increase
           * the size of the buffer used by Cilk_POLL() if you are
           * using that routine to receive PostClosure messages.
           */
     }
     CRITICAL_SECTION_END();
}

void PostClosureHandler(ClosureT *cp, int pn)
{
     assert(pn >= 0);
     assert(pn < cilk_active_size);

     CRITICAL_SECTION_BEGIN();

     if (pn == Self)
          Cilk_enqueue_handler(cp);
     else {
          cp->info = pn;
          /*
           * put here the code to send the closure to pn
           * and invoke post_rqsthdlr(cp->size). Don't forget to increase
           * the size of the buffer used by Cilk_POLL() if you are
           * using that routine to receive PostClosure messages.
           */
     }
     CRITICAL_SECTION_END();
}
```

Figure A.5: Skeleton of `PostClosureNode` and `PostClosureHandler`

```
static void HANDLER post_rqsthndlr(int size)
{
     ClosureT *newcp;

     newcp = (ClosureT *) Cilk_alloc_block(size);
     /*
      * put here the code to read the incoming closure into
      * newcp.
      */
     Cilk_AdoptClosure(newcp);
     PostClosureLocal(newcp);
}

static void HANDLER posthandler_rqsthndlr(int size)
{
     ClosureT *newcp;

     newcp = (ClosureT *) Cilk_alloc_block(size);
     /*
      * put here the code to read the incoming closure into
      * newcp.
      */
     Cilk_AdoptClosure(newcp);
     enqueue_handler(newcp);
}
```

Figure A.6: Skeleton of handlers for `PostClosureNode` and `PostClosureHandler`

Total execution time. In order to have a measurement of the execution time, you are required to edit `rts/conf.h`, enable the option `CILK_ELAPSED_TIME` and define two functions or macros in the architecture-specific files:

`void Cilk_TimerStart(void)` This function should start a machine-dependent timer (or record the current time somewhere).

`double Cilk_TimerStop(void)` This function returns the time (in seconds) elapsed since the last call to `Cilk_TimerStart()`.

Critical-path and work. Make sure that you have a very accurate clock on your machine (with a resolution of one μs or more) and make sure that reading it won't cost too much time (this operation is going to be executed millions of times). If the only way to read the clock is to issue a system call you are out of luck.

If your system satisfies these requirements, enable the option `CILK_TIMING` in the file `rts/conf.h`. Establish a time unit (for example a μs, or a CPU cycle) that is suited to your architecture. Cilk can handle whatever unit you choose, since it represents time as multiples of this unit, called a *cycle* in Cilk's jargon. Cilk's time is stored into a `unsigned long`: make sure that the execution time of a thread doesn't overflow the counter ($100\,\mathrm{ms}$ is a reasonable maximum time for a thread).

Now you you must provide the following macros or functions:

`unsigned long Cilk_CycleCount(void):` This function returns the current time, in cycles.

`unsigned long Cilk_ElapsedCycles(unsigned long t):` This function returns the number of cycles elapsed since time `t`.

`unsigned long Cilk_PackCycles(unsigned long n):` This function is used to collect the critical path timings, and its goal is twofold: I) to change the time scale so that the critical path time can still be held by an `unsigned long` without overflow (in practice it performs a division of `n` by some power of 2), and II) to adjust `n` to prevent errors. There is a problem on the CM5 (and possibly in other architectures), in that the system clock measures real-time; however the machine is timeshared, and a thread can be interrupted by the operating system. In this case the cycle counter measurse the execution time for a thread plus the time quantum in which another process has been executed. We therefore adopt the heuristic that a thread is considered broken if that thread ran for more than $50\,\mathrm{ms}$, and if a thread is broken, we give it a zero execution time.

`unsigned long Cilk_AdjustCycles(unsigned long n):` This macro must adjust `n` to prevent errors, but not change the time scale.

`double Cilk_UnpackCycles(unsigned long n):` Performs the opposite action, i.e. converts from packed cycles to cycles. The result is a `double`, since it could overflow a `long`.

`Cilk_CyclesToSeconds(n):` This macro (it should be a macro for technical reasons, since the type of `n` varies) converts from cycles into seconds. In practice it divides `n` by the number of cycles per second. The result must be cast to `double`.

A good testing program for this step is `testhandlers.c`: if you compile the program with `CILK_TIMING=1` it will invoke all the handlers and report the round-trip time measurements for all of them. As a comparison, the round-trip for sending a long value and get the answer is $\approx 35\,\mu\mathrm{s}$ on the CM-5.

Appendix B

Copyright and Disclaimers

Appendix C

Release Notes

C.1 Release 1.0 Beta2

The first public release.

C.2 Release 1.1 Beta1

Syntax changes over version 1.0 beta2. The statement form that spawns a thread is changed from

```
spawn (foo, args ...) stmt
```

to

```
spawn foo (args ...) stmt
```

The same changes apply to all cousins of `spawn`, i.e., `spawn_next`, `call`, `call_next`, `tail_call`, `tail_call_next`, `make_closure` and `make_next_closure`. (See the changed example programs.)

C.3 Release 1.2 Beta 1

- Fixed a bug where closures created with a zero-size accumulator were not being correctly posted.
- Added DoubleWord accumulators:

```
AccumulateDoubleWord (cont k, AccumDoubleWordOp op, long long delta);
accum_doubleword_add (doubleword value, doubleword *accum);
accum_doubleword_mul (doubleword value, doubleword *accum);
accum_doubleword_and (doubleword value, doubleword *accum);
accum_doubleword_or (doubleword value, doubleword *accum);
accum_doubleword_max (doubleword value, doubleword *accum);
accum_doubleword_min (doubleword value, doubleword *accum);
```

- Added `float` and `double` types to primitive communication operations. In particular, the following routines were added:

```
SendFloatArgument (cont k, float value);
SendDoubleArgument (cont k, double value);
AccumulateFloat (cont k, AccumFloatOp op, float delta);
AccumulateDouble (cont k, AccumDoubleOp op, double delta);
accum_float_add (float value, float *accum);
accum_float_mul (float value, float *accum);
accum_float_max (float value, float *accum);
accum_float_min (float value, float *accum);
accum_double_add (double value, double *accum);
accum_double_mul (double value, double *accum);
accum_double_max (double value, double *accum);
accum_double_min (double value, double *accum);
```

Appendix D

The Cilk Thread Preprocessor

NAME

cilkpp — The Cilk™ 1.2 language preprocessor

SYNOPSIS

cilkpp [-hvN] [-o *outfile* **]** *infile*

DESCRIPTION

cilkpp is the thread preprocessor for the Cilk™ 1.2 language as defined in *Cilk™ 1.2 Language Reference Manual*. It reads from an input Cilk program file, which by convention has the extension *.p*, and generates ordinary C code in the output file. The output file is named *c.out* unless explicitly specified. The generated C file must be compiled using *gcc* and linked with the Cilk runtime library.

The following options are available:

-v If the **-v** option is given, *cilkpp* just display a message showing the current version number and creation date.

-h If the **-h** option is given, *cilkpp* display a brief help message.

-N The **-N** option instructs *cilkpp* to suppress the generation of *cpp* `#line` directives. *cilkpp* inserts these directives into the generated C code by default, which enables C compiler diagnostic messages to be related to the Cilk source program.

With the **-N** option, line breaks are inserted into the generated C code so that it is human readable after being set by the UNIX *indent* program.

-o *outfile* The **-o** option instructs *cilkpp* to write the generated C code in *outfile*

SEE ALSO

Chapter 4 of the *Cilk™ 1.2 (Version β1) Reference Manual*.
MIMD Style Parallel Programming with Continuation-Passing Threads by M. Halbherr, Y. Zhou and C. Joerg.

BUGS

gcc is required to compile the C code output by *cilkpp*, as some of it are *gcc* extensions.

The line numbers are accurate only for one terminal symbol within each production in the grammar, therefore line numbers for some symbols such as '`)`', '}' or '`else`' may be slightly off.

Since different pieces of the C code generated by *cilkpp* may be derived from the same source, some errors in the source may get reported more than once from the C compiler.

Appendix E

Installation Instructions

Currently, these installation procedures only apply to the CM5. Hopefully, they will not need much modification for installing on other systems.

E.1 How to obtain Cilk

Cilk can be obtained by `ftp`ing the file `Cilk1.2.tar.Z` from
`theory.lcs.mit.edu:/pub/ftp/pub/cilk/Cilk1.2.tar.Z`
Cilk should be installed on the compile-server for the CM5.

E.2 How to Install Cilk

After copying `Cilk1.2.tar.Z` into a directory on your system, do the following things:

- type '`uncompress Cilk1.2.tar.Z`'. This will create a file named `Cilk1.2.tar`.
- type '`tar -xf Cilk1.2.tar`'. This will create a directory named `Cilk1.2/` and put all of the Cilk software into this directory. You may want to remove the `Cilk1.2.tar` file after this is done.
- type '`cd Cilk1.2`' followed by '`source INSTALL`'. This will install all of the Cilk software including the pre-processor, the run-time library, and the Strata communication library. Note that there are a few unaviodable compiler warnings when building Cilk, and these are listed in the README in the top directory.

Note: In order to compile the pre-processor, you need to have a tool called `flex`, the GNU version of the `lex` utility. Not all systems will have this software installed. The current distribution of `flex` can be found in any one of the GNU archive sites, for example `prep.ai.mit.edu` (in directory `pub/gnu`). You will also need `bison`, the GNU version of `Yacc`, as it is needed in building flex.

If you cannot get `flex`, an executable for the pre-processor for a sparc is provided. If you want to use this executable instead of building the pre-processor `cilkpp`, just comment out the three lines in the `INSTALL` script before running it:

```
pushd cilkpp
make cilkpp
popd
```

E.3 How to run the regression tests

E.3.1 Strata regression test

Move to the `Cilk1.2/strata` directory and type '`jrun do-test`'. This should run the strata regression test and tell you something about the raw performance of the CM5 you are using.

E.3.2 Cilk run-time system regression test

Move to the `Cilk1.2/rts` directory and type '`make testall_cm5`' to build the regression test program (this may take several minutes). Then edit the file `job` and uncomment the call to the regression test program (the line that says '`testall_cm5 0`'). Finally, type '`jrun job`' to run the regression test. The regression test program will print '`OK : ...`' for correctly functioning primitives and print '`BAD: ...`' for malfunctioning primitives. It may also hang if a primitive is seriously broken. The program may also print '`???:  ...`' if it is unable to determine whether a particular primitive is working or not.

E.3.3 Running the Cilk examples

Move to the `Cilk1.2/examples` directory and type '`make fib_cm5_st`'. This will make an executable for `fib` for the CM5 with statistics gathering enabled. Then edit the file `job` and uncomment the call to the `fib_cm5_st` program (the line that says '`fib_cm5_st 0 30`'). Finally, type '`jrun job`' to run the fib example. For the other examples (`queens`, `tree`, `adq`), simply repeat the above procedure with their name replacing `fib`.

E.4 Mailing Lists

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

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

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

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

Appendix F

Reporting Bugs

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

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

Or by hardcopy to

Cilk Bugs
c/o Bradley C. Kuszmaul
NE43-228
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Appendix G

Development Methodology

This appendix is written for the Cilk development team.

If you find a bug, and fix it, then make a regression test that demonstrates

- the absence of the bug in the new code, and
- the presence of the bug in old code.

Appendix H

Things To Do

- Add sample applications to the tutorial.
- Mention phish in the introduction.
- Add references. (Currently the references just has a list of papers, but nothing in the text of the manual refers to any of the papers. This was accomplished with `\nocite{*}`, but should have been done the right way.

Appendix I

Overall Change Log

Release 1.0 β 1 had two bugs in it: The cilkpp preprocesor failed to put curly braces in certain places; and when repeated calls to RunScheduler were made in AUTO or MAUNUAL mode, the run-time system would occasionally allocate the same block of memory for two different purposes.

Release 1.0 $beta2$ seems to fix those bugs.

```
$Log: manual.tex,v $
% Revision 1.45  1995/02/06  15:07:12  athena
% Added the implementor's guide.
%
% Revision 1.44  1994/11/06  02:19:39  randall
% Added AccumulateDoubleWord, SendDoubleArgument and AccumulateDouble to doc.
%
% Revision 1.43  1994/11/03  01:51:15  randall
% Updated everything to 1.2.  Added changes in 1.2 to the change log.
%
% Revision 1.42  1994/10/24  23:55:50  randall
% Updated ftp location of Cilk.
%
% Revision 1.41  1994/10/14  18:35:08  randall
% ARPA contract number update.
%
% Revision 1.40  1994/10/12  20:54:01  randall
% Updated title page footnote.
%
% Revision 1.39  1994/09/28  19:14:26  randall
% changed 1.0 -> 1.1
%
% Revision 1.38  1994/09/28  18:57:40  bradley
% Fix up credits.
%
% Revision 1.37  1994/09/28  18:21:53  zhou
% Add release notes as an appendix
```

```
%
% Revision 1.36  1994/09/23  03:17:22  bradley
% Add halbherr to author list.
%
% Revision 1.35  1994/09/08  22:33:31  bradley
% Add information about the bugs in 1.0 beta-1.
%
% Revision 1.34  1994/09/08  19:30:52  bradley
% Edit the disclaimer.
%
% Revision 1.33  1994/09/08  19:25:21  bradley
% gratuitous change
%
% Revision 1.32  1994/09/08  19:23:25  bradley
% Add trademark protection and up-to-date legal disclaimers.
%
% Revision 1.31  1994/09/07  17:26:17  bradley
% Update author list and remove annoying cvs "Id" from titlepage.
%
% Revision 1.30  1994/09/06  17:48:55  bradley
% Add references.
%
% Revision 1.29  1994/09/06  17:40:26  bradley
% Many small changes from my things-to-do list.
%
% Revision 1.28  1994/09/06  17:19:11  bradley
% Change silk to cilk
%
% Revision 1.27  1994/09/02  19:28:30  zhou
% Add instruction on how to obtain flex.
%
% Revision 1.26  1994/09/02  19:22:27  randall
% StrataInit -> CilkInit, same for exit.
%
% Revision 1.25  1994/09/02  18:24:21  randall
% Cilk->Cilk1.0, added ref for flex location.
%
% Revision 1.24  1994/09/02  01:30:51  zhou
% Changed tpp chapter to cilkpp
%
% Revision 1.23  1994/08/31  18:08:09  randall
% Added some to local guide, put note about flex into installation instructions.
%
% Revision 1.22  1994/08/30  20:44:16  zhou
% Add tpp manpage as an appendix
```

```
%
% Revision 1.21  1994/08/30  18:54:19  randall
% added some installation notes.
%
% Revision 1.20  1994/08/30  18:29:12  randall
% Updated Cilk.tar.Z location(s)
%
% Revision 1.19  1994/08/30  13:47:55  bradley
% Put the mailing list names in tt font.
%
% Revision 1.18  1994/08/30  13:47:19  bradley
% Change the some filenames in the documentation.
%
% Revision 1.17  1994/08/29  21:48:13  randall
% Put serial fib ahead of threaded fib in tutorial.
% Changed some local guide stuff and how to obtain Cilk stuff.
%
% Revision 1.16  1994/08/29  21:10:16  randall
% Updated installation instructions.  Added disclaimer & copyright.
%
% Revision 1.15  1994/08/29  15:18:59  zhou
% Added a section to explain tpp output.
%
% Revision 1.14  1994/08/26  18:17:06  randall
% Added table of contents.  Added some installation instructions.
%
% Revision 1.13  1994/08/26  18:00:23  bradley
% added mailing list info.
%
% Revision 1.12  1994/08/26  15:41:49  bradley
% Filled in the bug information.
%
% Revision 1.11  1994/08/26  15:04:13  zhou
% Add section for data primitives
%
% Revision 1.10  1994/08/25  18:01:11  bradley
% Fixed author list, and added a few changes.
%
% Revision 1.9  1994/08/25  17:34:47  randall
% Finished some sections in tutorial, added some information to the
% local guide to tell where Cilk software is located.
%
% Revision 1.8  1994/08/25  14:55:33  randall
% Made all of the undefined \ref{}s work.
%
```

```
% Revision 1.7  1994/08/24  20:34:54  randall
% Made tc-tut compile into manual.
%
% Revision 1.6  1994/08/24  15:23:17  bradley
% Added a changelog to langref.tex
% changed mac.tex to defs.tex.
%
% Revision 1.5  1994/08/24  15:11:28  zhou
% added lanref + mac.
%
% Revision 1.4  1994/08/22  16:38:48  bradley
% Added local guide.
%
% Revision 1.3  1994/08/22  16:29:07  bradley
% An initial version of the manual with some stuff in it.
%
% Revision 1.2  1994/08/22  16:18:42  bradley
% Almost the first version.
%
```