Lecture 8: Cilk Language and Implementation

1 Implementation of Cilk-1...Cilk-5

Consider a well-known Cilk program:

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

What is going on underneath?

The way Cilk executes the program changed radically from Cilk 1 to Cilk 4.

1.1 Cilk 1 Implementation

In Cilk 1, each thread t is a separate C function with one argument and the return type of *void*. The lone argument is a pointer to a run-time data structure called a *closure* which holds the arguments for t.

```
struct Closure {
    Pointer to the C function for thread t
    Slot for each argument
    Join Counter - /* indicates the # of missing arguments before
                      the thread can execute. */
}
```

When we spawn threads, it's important to know whether there are any arguments to be filled in. We use the join counter in the closure to prevent threads from being exectued before the arguments are ready.

For scheduling, we can use the join counter as follows:

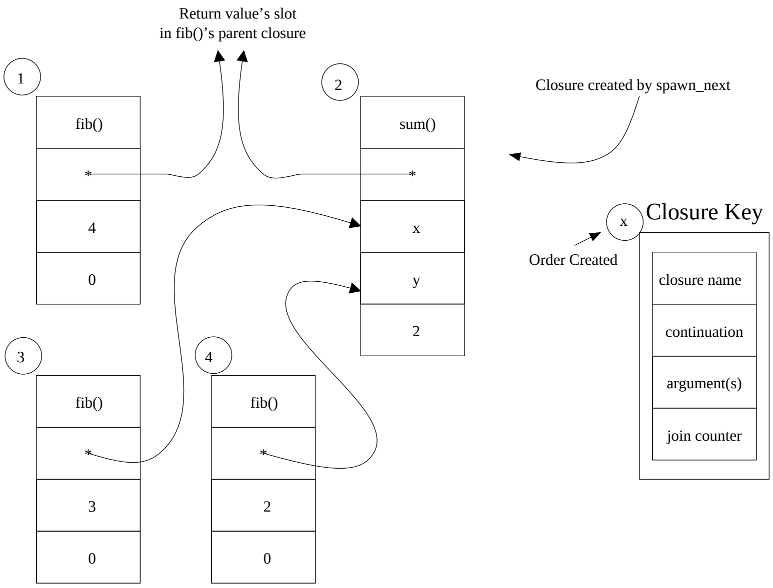

Figure 1: Explicit Continuation Passing

- A closure with join counter > 0 must wait for arguments.
- A closure with join counter == 0 is ready for to execute.

We have yet to describe how are arguments are filled in and the join counter is decremented. For this, we introduce *continuations*. A continuation is a pointer to an argument slot, with an offset to the join counter. `send_argument(k,x)` may then be implemented to place a value x into a slot specified by continuation k and to decrement the join counter of the closure that k resides in.

Instead of join counters, some multithreaded languages use a presence bit for each argument to represent whether the argument in a given slot is ready. The advantage of presence bits is that `send_argument()` can use the presence bits to detect overwriting. The disadvantage is that presence bits may require more space and time to implement when the number of arguments in a procedure exceeds the number of bits in the processor's word size.

1.1.1 Explicit Continuation Passing

1.1.2 Cilk-1 Example

```
thread fib(cont int k, int n)
{
  if (n < 2)
    send_argument(k,n);
  else {
    cont int x,y;
```

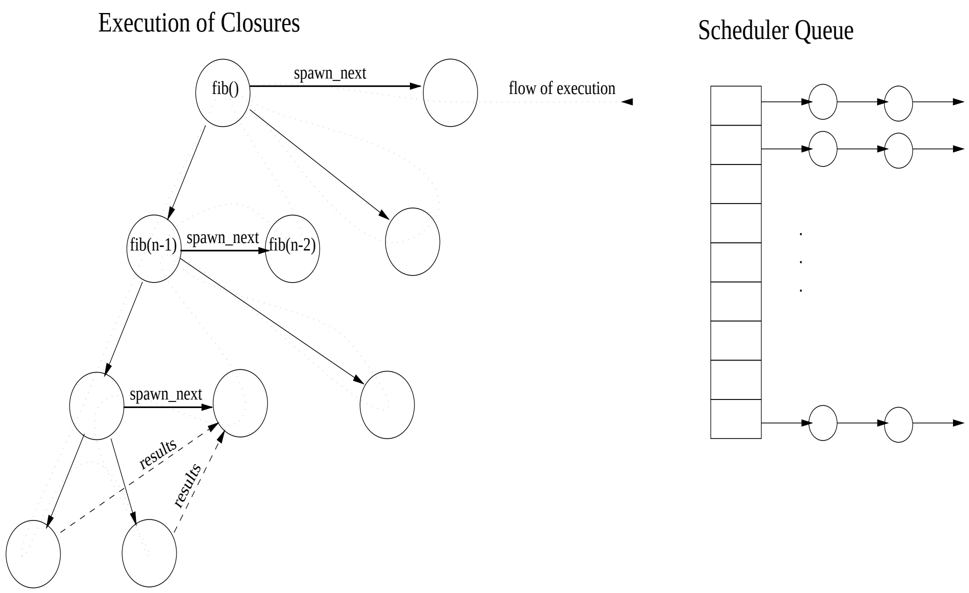

Figure 2: Serial Execution of Cilk-1 and Cilk-2

```
    /* Spawn next => Spawning next part of this procedure */
    spawn_next sum(k, ?x, ?y); /* '?' => Continuation */
    spawn fib(x,n-1); /* will fill in continuation of x */
    spawn fib(y,n-2); /* will fill in continuation of y */
  }
}

thread sum(cont int k, int x, int y) /* continuation of fib procedure */
{
  send_argument(k,x+y); /* fills in continuation of fib's caller */
}
```

In the Cilk-1 language, the construct `?x` captures the pointer to the slot in the closure and stores it in the variable `x`. For example, after `spawn_next sum(k, ?x, ?y)` is executed, `x` contains a pointer to the third slot in the closure of `sum`.

1.2 Cilk-2

Cilk-2 compiles high level cilk code with `spawn`s and `sync`s to lower level Cilk code, as shown above, with `spawn_next()`s, `thread`s and continuations.

Cilk-1 and Cilk-2 are not quite C-like in that during execution every thread specified as being spawned before the first `sync` is spawned by the Cilk environment before any such thread is executed. Hence, the execution is especially not C-like with respect to space and other aspects of stack behavior. For example:

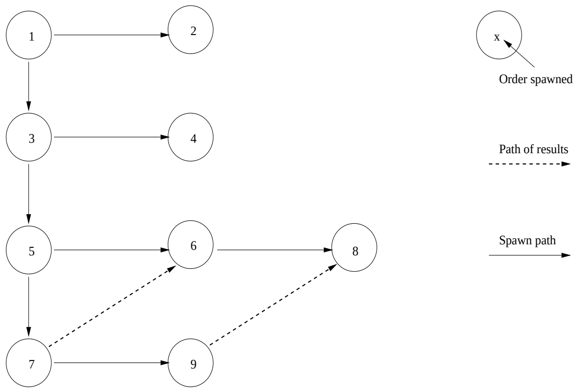

Figure 3: Lazy Task Creation

```
for(i=0; i< 1000000; i++)
  spawn foo();
```

The above code will spawn 1,000,000 closures before any `foo()` executes.

1.3 Cilk-4

Cilk-4 implemented the *lazy task creation* idea of *Mohr, Kranz, Halstead.* Using this algorithm, a procedure spawns each child only when it is ready to execute that child. (sounds biblical, doesn't it?)

	For K Children Spawned	*Faster?*
Cilk-1	K + 1 context Switches	If context switch is greatest cost
Cilk-4	2K Context Switches	If space or locality are factors

Cilk-4's lazy task creation algorithm requires more context switches, but tends to be faster than the spawn-all algorithm of Cilk-1 because modern processors are optimized for the stack-like behavior Cilk-4's algorithm. Increased performance as a result of lazy task creation is also seen because stack frames can be re-used more often, saving space and increasing locality.

1.4 Cilk-5

Cilk-5's approach is an hybrid between Cilk-1 and Cilk-4. The scheduler runs in lazy-task mode most of the time (i.e., keeping only partial information in a stack-like fashion). However, it can also run in Cilk-1 mode (keeping complete information about closures, connected in a more-or-less arbitrary dag). The advantage is that Cilk-5 can implement advanced features (like `abort`) without altering the scheduler too much. Performance is not sacrificed in the common case.

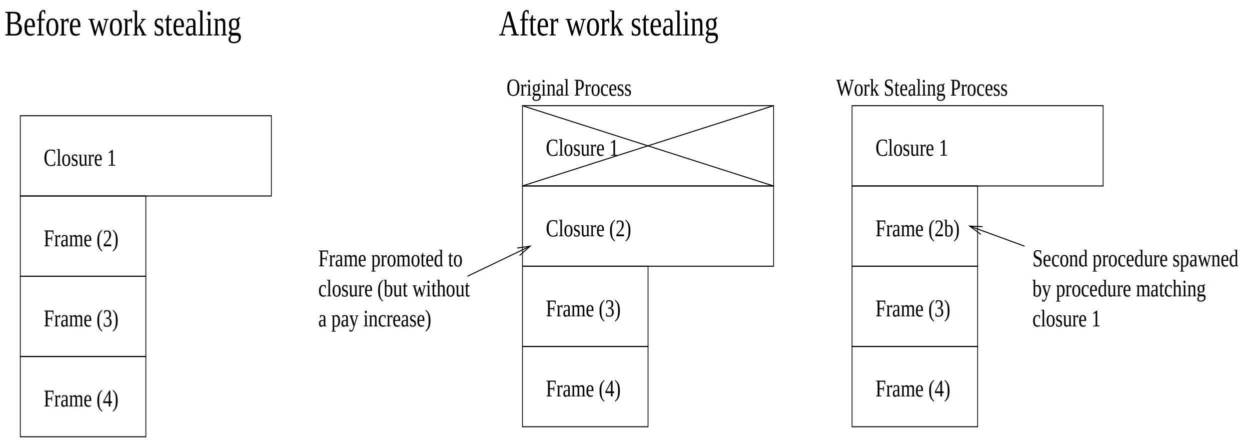

Figure 4: Closure Stealing

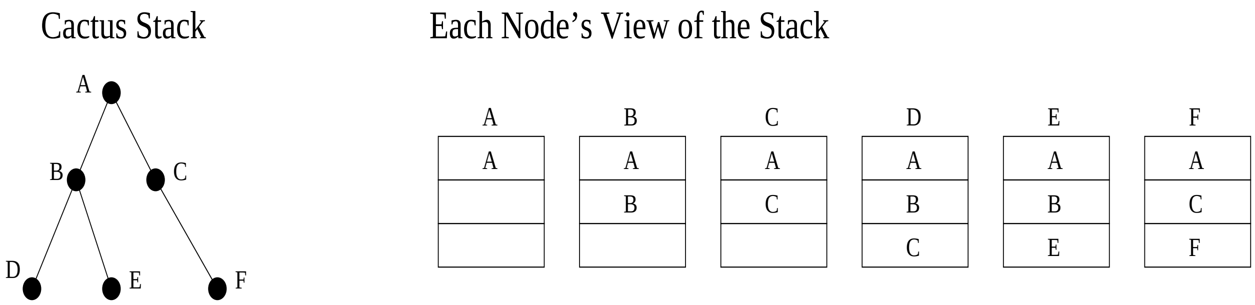

Figure 5: Cactus Stack

1.4.1 Work Stealing

When the top closure is stolen from the top, the stack frame below it is promoted to a closure.

1.4.2 Memory Management

In C, one should never return as a function result a pointer to any variable within the procedure's stack frame. If one did, the memory the result pointer referenced would become invalid as the stack frame was freed, and before the result was received by the caller. While it is valid to return memory allocated from the heap using `malloc()`, such memory must be explicitly freed.

This view of memory is what is known as a *cactus stack*, where a procedure may only see variables belonging to itself and it's parent procedures.

Cilk has the same rules for pointers as does C. You may always pass pointers downward on the stack, but you may only pass pointers to heap storage upward.

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

  /*****************************************************
  ** Both of the below instances of bar() may read x safely.
  ** If an instance of bar should try to write, a race
  ** condition will occur.
  *****************************************************/
  spawn bar(&x);
  spawn bar(&x);

  sync;
  return x;
}

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

The above example shows valid Cilk code that exhibits nondeterminism. Cilk-5 provides a tool for checking for race conditions. It's called the Nondeterminator.

2 Inlets

Common case of spawning - results are stored into variables:

`x = spawn foo(y,z);`

Sometimes function results are used to update, rather than overwrite, a variable:

`x+= spawn foo(y,z);`

How should the update be done?

1. Child does the update
   - Slows child
   - Adds code to child
2. Parent does the update
   - Parent usually busy doing something else
   - The child must then pass value and interrupt parent
3. Inlet

- Create a Cilk-1 style closure for the return value
- Closure contains code for doing update
- Execute closure when argument (foo's return value) is available.
- Bonus! - Within inlets you can kill caller's children (preventing the children's processors from doing unnecessary work.)

```
cilk int fib(int n)
{
  int x = 0;
  inlet void summer(int result)
  {
    x += result;
    return;
  }
  if ( n < 2 ) {
    return n;
  } else {
    summer( spawn fib(n-1) );
    summer( spawn fib(n-2) );
    sync;
    return x;
  }
}
```

Looks like a lot of work? Well, Cilk will create inlets for you. The special syntax

```
x += spawn fib(n-1);
```

will create the inlet automatically.