Lecture 10a: Cilk and Work-Stealing Schedulers

Sergei Uversky

April 03, 2012

Abstract

In previous lectures, we considered the implications of using thread-spawning systems such as POSIX or Erlang in order to implement concurrency. In this lecture, however, we instead look at an alternative approach to concurrency, which utilizes tasks and lazy task creation in order to parallelize programs.

1 Introduction to Tasks

Previously in this course, we have considered systems such as POSIX and Erlang, which have allowed us to implement concurrency via thread creation. In such a system, the creation of a thread inherently evokes a large amount of work (e.g. space allocation), which has to be done by a scheduler. Thus, working in such a system inherently involves balancing the creation of these threads with optimization and fine-tuning, in order to mitigate performance loss.

However, it is possible to conceive of a different approach to concurrent programming; one which avoids the overhead native to the aforementioned systems. For example, consider a generic `thread_create( f() )` function. In the previous context, this function is seen as a command to spawn a thread no matter what; however, in this task-oriented scenario, we will instead view it as a "hint" to create a thread - i.e., if there are enough resources available to make a thread, then we shall do so, but otherwise we will simply run the function sequentially. We shall call this "hint" function `make_thread?( f() )`.

This naive approach, regrettably, is not without its flaws. Let us consider some of them:

- Given n cores, if we attempt to run a number of processes $P(k)$ concurrently, the $n + 1^{th}$ process and onwards will inevitably run in the same thread. However, what if $P(1...n)$ execute and return quickly, whereas $P(n+1...k)$ do not? The earlier cores will then be unutilized, creating an unnecessary drop in performance!
- The "sequentialized" processes created after the first n might generate data used by other processes, creating unnecessary overhead and defeating the purpose of the concurrent paradigm we studied at the beginning of the semester.

- The means by which we check if resources are available might be resource-intensive themselves - however, this is not necessarily true, as the resource-checking procedure is completely abstract.

Despite these limitations, there are ways to utilize this approach and still yield notable performance gains. Let us consider the pseudocode for a simple function `sum_tree(tree)`, whose purpose is to sum the leaves of a tree. The code is shown below, in Figure 1.

```
1.   function sum_tree(tree) = {
2.       if leaf?(tree) then
3.           leaf_value(tree)
4.       else
5.           sum_tree(left(tree)) + sum_tree(right(tree))
6.   }
```

Figure 1: Sum_tree function.

`sum_tree(tree)` is easily parallelizable. We can imagine "marking" the left recursion in line 5 as a point where we want to create another thread, and have the right recursion stay in the "main" thread. We thus get a system of threads branching left, as seen below:

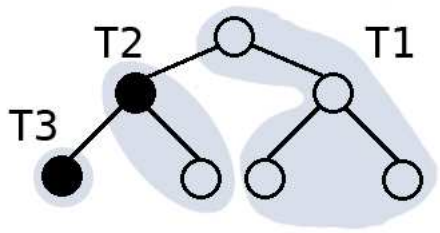

Figure 2: Left tree thread branching.

Note, however, that this parallelized system grows not with our resources or number of cores n, but instead with the size of the j-element tree, or simply $log(j)$. This is obviously problematic, as it falls into all of the pitfalls discussed before.

Now, what if we were able to take a somewhat different approach in which we lazily create threads, holding off on creation until we have reached the point where we *know* that creating a thread will help us? This is the crux of the approach shown in Figure 3 on the following page, dubbed *lazy task creation*:

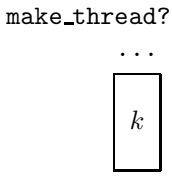

Figure 3: Lazy task creation with continuations.

Looking back to our knowledge of continuations, it is clear that there are two important elements to consider when looking at the abstract creation of a thread - the thread itself, and the continuation that follows the thread. Instead of explicitly creating a thread (which is not necessarily needed), we should consider a different approach:

1. Have the program begin running `f()`, and
2. Have the program say that there is some continuation k which another processor is now free to execute.

Consider an environment with functions `f()`, `g()` and `h()` and continuations k_f, k_g and k_h respective to the preceding functions. If a processor were to run `f()`, after `f()` returns, it would attempt to run k_f (the continuation associated with `f()`). If k_f has been "taken" already - that is, if another processor has begun executing k_f while the first was busy executing `f()` - then the processor will signal other processors that it is free to perform the other continuations.

In such a system, we are creating *tasks* - potential creation points of threads. The preceding example actually demonstrated *lazy task creation*!

2 Cilk

The language Cilk[1] (which is based on C) is founded on the concept of lazy task creation, and utilizes continuations as the fundamental unit of processing. Cilk is efficient in that it does not allocate stack space, save registers, etc. (all of the typical overhead associated with threading) until we know that we can do so. In this lecture and the following, we will look at the structure of Cilk programs and its programming philosophy.

Cilk is somewhat counterintuitive, in two main senses:

1. Cilk is optimized primarily for serial execution. This seems especially strange, but some careful thinking about Amdahl's law shows that this is the best way to improve performance - after all, the primary source of bottlenecking in practical concurrent applications is the fraction of code which is not parallelizable!

[1] Intel's Thread Building Blocks are primarily based on Cilk, in order to improve performance.

2. Cilk inverts expectations of what will be executed remotely. The Cilk equivalent of `make_thread?( f() )` (our abstract task spawner) actually executes `f()` locally and the *continuation* k_f is what has the potential to be executed remotely.

Consider the following Cilk program, which takes a basic recursive approach to calculating the n^{th} Fibonacci number:

```
1.    cilk int fib (int n) {
2.        int n1, n2;
3.        if (n < 2)
4.            return n;
5.        else {
6.            n1 = spawn fib(n - 1);
7.            n2 = spawn fib(n - 2);
8.            sync;
9.            return (n1 + n2);
10.       }
11.   }
```

Figure 4: Fibonacci function in Cilk.

Note the use of the `cilk` header (indicating the function is a Cilk function), and the associated keywords `spawn` and `sync`. The `spawn` keyword embodies the lazy task creation approach discussed earlier, and allows the given procedures (in this case, `fib(n - 1)` and `fib(n - 2)`) to be executed asynchronously with the caller. The The `sync` keyword is the "join point", where the current thread waits for all locally-spawned tasks to return before continuing. In Cilk, procedures cannot terminate while they have outstanding children (i.e. children which have spawned tasks which have not yet returned).

Cilk is very faithful to C - indeed, a properly written Cilk program will turn into a C program with acceptable, serial execution if all Cilk keywords are erased. This is extended to function calls - the `spawn` keyword can only be applied to Cilk functions, and Cilk functions cannot be called by normal means, instead requiring a `spawn`. Thus, in a Cilk program, `main()` must be declared as a Cilk function, in order to allow parallelization of the rest of the code.

We have had to be rather careful with our terminology here, as a thread in Cilk has a very particular definition:

thread The maximal sequence of instructions not containing a spawn, sync or return (that is, a unit of work that can actually have some sort of resources assigned to it).

The lecture concluded with a look at a model of Cilk execution, which gives a graphical representation of the concepts we discussed the rest of the day. The

model is shown below, with circles showing threads, downward edges showing the spawning of a new subcomputation, and horizontal edges showing the transfer of a continuation to a successor thread. Upward edges indicate the return of a value to a parent.

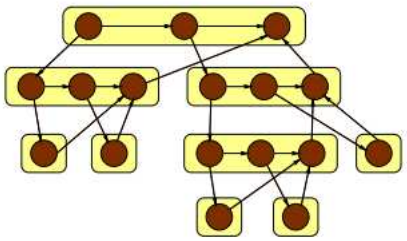

Figure 5: Cilk execution model.

The following lecture yields a more in-depth look into the structure and concepts behind Cilk programs.