ECE459: Programming for Performance Winter 2011

Lecture 9 — February 1, 2011

Patrick Lam *version 1*

Breaking Dependencies with Speculation

Recall that computer architects often use speculation to predict branch targets: the direction of the branch depends on the condition codes when executing the branch code. To get around having to wait, the processor speculatively executes one of the branch targets, and cleans up if it has to.

We can also use speculation at a coarser-grained level and speculatively parallelize code. We discuss two ways of doing so: one which we'll call speculative execution, the other value speculation.

Speculative Execution for Threads.

The idea here is to start up a thread to compute a result that you may or may not need. Consider the following code:

```
void doWork(int x, int y) {
  int value = longCalculation(x, y);
  if (value > threshold) {
    return value + secondLongCalculation(x, y);
  }
  else {
    return value;
  }
}
```

Without more information, you don't know whether you'll have to execute `secondLongCalculation` or not; it depends on the return value of `longCalculation`.

Fortunately, the arguments to `secondLongCalculation` do not depend on `longCalculation`, so we can call it at any point. Here's one way to speculatively thread the work:

```
void doWork(int x, int y) {
  int value1, value2;
  #pragma start parallel region
  {
    #pragma start parallel task
    {
      value1 = longCalculation(x, y);
    }
    #pragma start parallel task
    {
      value2 = secondLongCalculation(x, y);
```

```
    }
  }
  #pragma wait for parallel tasks to complete
  if (value1 > threshold) {
    return value1 + value2;
  }
  else {
    return value1;
  }
}
```

We now execute both of the calculations in parallel and return the same result as before.

Intuitively: when is this code faster? When is it slower? How could you improve the use of threads?

We can model the above code by estimating the probability p that the second calculation needs to run, the time T_1 that it takes to run `longCalculation`, the time T_2 that it takes to run `secondLongCalculation`, and synchronization overhead S. Then the original code takes time

$$T = T_1 + pT_2,$$

while the speculative code takes time

$$T_s = \max(T_1, T_2) + S.$$

Exercise. Symbolically compute when it's profitable to do the speculation as shown above. There are two cases: $T_1 > T_2$ and $T_1 < T_2$. (You can ignore $T_1 = T_2$.)

Value Speculation

The other kind of speculation is value speculation. In this case, there is a (true) dependency between the result of a computation and its successor:

```
void doWork(int x, int y) {
  int value = longCalculation(x, y);
  return secondLongCalculation(value);
}
```

If the result of `value` is predictable, then we can speculatively execute `secondLongCalculation` based on the predicted value. (Most values in programs are indeed predictable).

```
void doWork(int x, int y) {
  int value1, value2;
  static int last_value;
```

```
  #pragma start parallel region
  {
    #pragma perform parallel task
    {
      value1 = longCalculation(x, y);
    }
    #pragma perform parallel task
    {
      value2 = secondLongCalculation(last_value);
    }
  }
  #pragma wait for parallel tasks to complete
  if (value1 == last_value) {
    return value2;
  } else {
      last_value = value1;
      return secondLongCalculation(value1);
  }
}
```

Note that this is somewhat similar to memoization, except with parallelization thrown in. In this case, the original running time is

$$T = T_1 + T_2,$$

while the speculatively parallelized code takes time

$$T_s = \max(T_1, T_2) + S + pT_2,$$

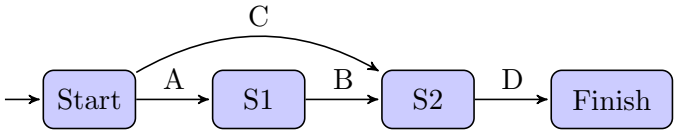
where S and p are the same as above.

Exercise. Do the same computation as for speculative execution.

Critical Paths

You should be familiar with the concept of a critical path from previous courses; it is the minimum amount of time to complete the task, taking dependencies into account, and without speculating.

Consider the following diagram, which illustrates dependencies between tasks (shown on the arrows). Note that B depends on A, and D depends on B and C, but C does not depend on anything, so it could be done in parallel with everything else. You can also compute expected execution times for different strategies.



How to Parallelize Code

Here's a four-step outline of what you need to do.

1. Profile the code.
2. Find dependencies in hotspots. For each dependency chain in a hotspot, figure out if you can execute the chain as multiple parallel tasks or a loop over multiple parallel iterations. Think about changing the algorithm, if that would help.
3. Estimate benefits.
4. If they're not good enough (e.g. far from linear speedup), step back and see if you can parallelize something else up the call chain, or at a higher level of absraction. (Think Mandelbrot sets and computing different points in parallel).

Try to reduce the amuont of synchronization that you have to do (waiting for parallel tasks to finish), because that always slows you down.

Creating and Using Threads

Here's a quick `pthreads` refresher. To compile a C or C++ program with pthreads, add the `-pthread` parameter to the compiler commandline on Linux. (You are on your own on Windows.) Then, you can start a thread with the call `pthread_create()` as follows:

```
#include <pthread.h>

...
{
  thread_info ti;
  int s = pthread_create(&ti,
                         NULL, /* optional attributes */
                         &thread_main,
                         "thread arguments");

  // if you want to wait for the thread to finish and collect the
  // return value:
  s = pthread_join(ti, &thread_return_value);
}

void * thread_main(void * arg) {
  /* put thread body here */
  pthread_exit(status);
  // or, you can just return from the thread:
  return status;
}
```

Thread Pools. We talked about "single task, multiple threads" last week. The idea behind a *thread pool* is that it's relatively expensive to start a thread; it costs resources to keep the threads running at the operating system level; and the threads won't run optimally anyway, because they'll spend too much time swapping state in and out of the cache. Instead, you start an appropriate number of threads, which each grab work from a work queue, do the work, and report the results back. Web servers are a good application of thread pools[1].

A key question is: how many threads should you create? This depends on which resources your threads use; if you are writing computationally-intensive threads, then you probably want to have fewer threads than the number of virtual CPUs. You can also use Amdahl's Law to estimate the maximum useful number of threads, as discussed previously.

Here's a longer discussion of thread pools:

`http://www.ibm.com/developerworks/library/j-jtp0730.html`

Modern languages provide thread pools; Java's `java.util.concurrent.ThreadPoolExecutor`[2], C#'s `System.Threading.ThreadPool`[3], and GLib's `GThreadPool`[4] all implement thread pools.

GLib. GLib is a C library developed by the GTK team. It provides many useful features that you might otherwise have to implement yourself in C. Consider using GLib's thread pool in your assignment 2, unless you want to implement the work queue yourself. (I see no reason to do that!)

Thread-related Complications

There are a number of complications which arise when you use threads. One of the classic ones is *data races*: multiple threads attempt to access the same data, with at least one of the accesses being a write. Data races do result in nondeterministic behaviour, so they're usually considered bad. (Not all data races are actually harmful, though.) One way to control data races is by ensuring that accesses to shared data are protected by locks. We'll expand on this later in a few weeks.

[1] Apache does this: `http://httpd.apache.org/docs/2.0/mod/worker.html`. Also see an assignment where the students write thread-pooled web servers: `http://www.cse.nd.edu/~dthain/courses/cse30341/spring2009/project4/project4.html`.

[2] `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html`

[3] `http://msdn.microsoft.com/en-us/library/3dasc8as\%28v=vs.80\%29.aspx`

[4] `http://library.gnome.org/devel/glib/unstable/glib-Thread-Pools.html#GThreadPool`