ECE459: Programming for Performance Winter 2015

Lecture 11 — January 28, 2015

Patrick Lam *version 1*

We talked about C++11 atomics last time. Someone asked about whether there was a Pthreads equivalent. Nope, not really.

`gcc` supports atomics via extensions:
`https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`

OS X has atomics via OS calls:
`https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html`

etc...

Reference: `http://stackoverflow.com/questions/1130018/unix-portable-atomic-operations`

Dependencies

I've said that some computations appear to be "inherently sequential". We talked about a bunch of real-life analogies:

- must extract bicycle from garage before closing garage door
- must close washing machine door before starting the cycle
- must be called on before answering questions? (sort of)
- students must submit assignment before course staff can mark the assignment

Note that, in this lecture, we are going to assume that memory accesses follow the sequentially consistent memory model. For instance, if you declared all variables to be C++11 atomics, that would be fine. This reasoning is not guaranteed to work in the presence of undefined behaviour, which exists when you have data races.

Main Idea. A *dependency* prevents parallelization when the computation XY produces a different result from the computation YX.

Loop- and Memory-Carried Dependencies. We distinguish between *loop-carried* and *memory-carried* dependencies. In a loop-carried dependency, an iteration depends on the result of the previous iteration. For instance, consider this code to compute whether a complex number $x_0 + iy_0$ belongs to the Mandelbrot set.

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

In this case, it's impossible to parallelize loop iterations, because each iteration *depends* on the (x, y) values calculated in the previous iteration. For any particular $x_0 + iy_0$, you have to run the loop iterations sequentially.

Note that you can parallelize the Mandelbrot set calculation by computing the result simultaneously over many points at once. Indeed, that is a classic "embarassingly parallel" problem, because the you can compute the result for all of the points simultaneously, with no need to communicate.

On the other hand, a memory-carried dependency is one where the result of a computation *depends* on the order in which two memory accesses occur. For instance:

```
int val = 0;

void g() { val = 1; }
void h() { val = val + 2; }
```

What are the possible outcomes after executing `g()` and `h()` in parallel threads?

RAW, WAR, WAW and RAR

The most obvious case of a dependency is as follows:

```
int y = f(x);
int z = g(y);
```

This is a read-after-write (RAW), or "true" dependency: the first statement writes `y` and the second statement reads it. Other types of dependencies are:

	Read 2nd	**Write 2nd**
Read 1st	Read after read (RAR)	Write after read (WAR)
	No dependency	Antidependency
Write 1st	Read after write (RAW)	Write after write (WAW)
	True dependency	Output dependency

The no-dependency case (RAR) is clear. Declaring data immutable in your program is a good way to ensure no dependencies.

Let's look at an antidependency (WAR) example.

```
void antiDependency(int z) {
  int y = f(x);
  x = z + 1;
}
```

```
void fixedAntiDependency(int z) {
  int x_copy = x;
  int y = f(x_copy);
  x = z + 1;
}
```

Why is there a problem?

Finally, WAWs can also inhibit parallelization:

```
void outputDependency(int x, int z) {
  y = x + 1;
  y = z + 1;
}
```

```
void fixedOutputDependency(int x, int z) {
  y_copy = x + 1;
  y = z + 1;
}
```

In both of these cases, renaming or copying data can eliminate the dependence and enable parallelization. Of course, copying data also takes time and uses cache, so it's not free. One might change the output locations of both statements and then copy in the correct output. These are usually more useful when it's not just one access, but some sort of longer computation.

Loop-carried Dependencies

As we said last time, a loop-carried dependency is one where an iteration depends on the result of the previous iteration. Let's look at a couple of examples.

Example 1 *Initially,* `a[0]` *and* `a[1]` *are 1. Can we run these lines in parallel?*

```
a[4] = a[0] + 1;
a[5] = a[1] + 2;
```

`http://www.youtube.com/watch?v=jjXyqcx-mYY`. (This one is legit! Really!)

It turns out that there are no dependencies between the two lines. But this is an atypical use of arrays. Let's look at more typical uses.

Example 2 *What about this? (Again, all elements initially 1.)*

```
for (int i = 1; i < 12; ++i)
    a[i] = a[i-1] + 1;
```

Nope! We can unroll the first two iterations:

```
a[1] = a[0] + 1
a[2] = a[1] + 1
```

Depending on the execution order, either `a[2] = 3` or `a[2] = 2`. In fact, no out-of-order execution here is safe—statements depend on previous loop iterations, which exemplifies the notion of a *loop-carried dependency*. You would have to play more complicated games to parallelize this.

Example 3 *Now consider this example—is it parallelizable? (Again, all elements initially 1.)*

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1;
```

Yes, to a degree. We can execute 4 statements in parallel at a time:

- a[4] = a[0] + 1, a[8] = a[4] + 1
- a[5] = a[1] + 1, a[9] = a[5] + 1
- a[6] = a[2] + 1, a[10] = a[6] + 1
- a[7] = a[3] + 1, a[11] = a[7] + 1

We can say that the array accesses have stride 4—there are no dependencies between adjacent array elements. In general, consider dependencies between iterations.

Larger loop-carried dependency example. Now consider the following function.

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

How do we parallelize this?

Well, that's a trick question. There's not much that you can do with that function. What you can do is to run this function sequentially for each point, and parallelize along the different points.

As mentioned in class, but one potential problem with that approach is that one point may take disproportionately long. The safe thing to do is to parcel out the work at a finer granularity. There are (unsafe!) techniques for dealing with that too. We'll talk about that later.

What I did do in class was to actually live-code a parallelization of the Mandelbrot code. I had to first refactor the code, creating an array to hold the output. Then I added a struct to pass the offset and stride to the thread. Finally, I invoked pthread to create and join threads.