ECE459: Programming for Performance Winter 2015

Lecture 11 — January 28, 2015

Patrick Lam *version 0*

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". Here's why.

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.