ECE459: Programming for Performance Winter 2014

Lecture 8 — January 30, 2014

Patrick Lam *version 1*

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.

I didn't mention it in class, but one potential problem with that approach is that one point may take disproportionately long. 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.

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) {
  thread_t t1, t2;
  point p(x,y);
  int v1, v2;
  thread_create(&t1, NULL, &longCalculation, &p);
  thread_create(&t2, NULL, &secondLongCalculation, &p);
  thread_join(t1, &v1);
  thread_join(t2, &v2);
  if (v1 > threshold) {
    return v1 + v2;
  } else {
    return v1;
  }
}
```

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) {
    thread_t t1, t2;
    point p(x,y);
    int v1, v2, last_value;
    thread_create(&t1, NULL, &longCalculation, &p);
    thread_create(&t2, NULL, &secondLongCalculation,
                  &last_value);
    thread_join(t1, &v1);
    thread_join(t2, &v2);
    if (v1 == last_value) {
      return v2;
    } else {
      last_value = v1;
      return secondLongCalculation(v1);
    }
}
```

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.

When can we speculate?

Speculation isn't always safe. We need the following conditions:

- `longCalculation` and `secondLongCalculation` must not call each other.
- `secondLongCalculation` must not depend on any values set or modified by `longCalculation`.
- The return value of `longCalculation` must be deterministic.

As a general warning: Consider the *side effects* of function calls.