

Lecture 16 — Dependencies and Speculation

Patrick Lam and Jeff Zarnett

2018-12-15

Dependencies

Some computations appear to be “inherently sequential”. There are plenty 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, some people shout out...)
- students must submit assignment before course staff can mark the assignment (also sort of... I can assign you a grade of zero if you didn't submit an assignment!)

There are some prerequisite steps that need to be taken before a given step can take place. The problem is that we need some result or state from an earlier step before we can go on to the next step. Interestingly, in many of the analogies, sometimes if you fail to respect the dependency, nothing physically stops the next step from taking place, but the outcome might not be what you want (... you don't want zero on your assignment, right?).

The same with dependencies in computation. If you need the result of the last step you will have to wait for it to be available before you can go on to the next. And if you jump the gun and try to do it early, you will get the wrong result (if any at all).

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 “embarrassingly 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?

Loop-carried Dependencies

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.

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.

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.

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.

Breaking Dependencies with Speculation

Let's go back to a real life analogy of speculation. Under normal circumstances, the coffee shop staff waits for you to place your order ("medium double double") before they start making your order. Sensible. If you go to a certain coffee shop enough, then the staff start to know you and know your typical order and they might speculate about your order and start preparing it in advance, even before you get up to the counter. If they're right, time is saved: your order is ready sooner. If they're wrong, the staff did some unnecessary work and they'll throw away that result and start again with what you did order. If they can predict with high accuracy what you want, then most of the time this is a benefit, and that's what we want.

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 is still the synchronization overhead, and p is the probability that `v1 != last_value`.

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. Oh, let's talk about side effects. Why not. They have a big impact on parallelism. Side effects are problematic, but why? For one thing they're kind of unpredictable (why does calling this function result in unexpected changes elsewhere?!). Side effects are changes in state that do not depend on the function input. Calling a function or expression has a side effect if it has some visible effect on the outside world. Some things necessarily have side effects, like printing to the console. Others are side effects which may be avoidable if we can help it, like modifying a global variable.

Code that allows multiple concurrent invocations without affecting the outcome is called *reentrant* or "pure" (and the use of the word pure shouldn't imply any sort of moral judgement on the code). It is a desirable property to have code that is reentrant if we want to parallelize things. If a function is not reentrant, it may not be possible to make it thread safe. And furthermore, a reentrant function cannot call a non-reentrant one (and maintain its status as reentrant).

Side effects are sort of undesirable, but not necessarily bad. Printing to console is unavoidably making use of a side effect, but it's what we want. Nevertheless, it should be obvious that when printing we can't have reentrant behaviour because two threads trying to write at the same time to the console would result in jumbled output. Or alternatively, restarting the print routine might result in some doubled characters on the screen.

The trivial example of a non-reentrant C function:

```

int tmp;

void swap( int x, int y ) {
    tmp = y;
    y = x;
    x = tmp;
}

```

Why is this non-reentrant? Because there is a global variable `tmp` and it is changed on every invocation of the function. We can make the code reentrant by moving the declaration of `tmp` inside the function, which would mean that every invocation is independent of every other. And thus it would be thread safe, too.

Remember that in things like interrupt subroutines (ISRs) having the code be reentrant is very important. Interrupts can get interrupted by higher priority interrupts and when that happens the ISR may simply be restarted (or we're going to break off handling what we're doing and call the same ISR in the middle of the current one). Either way, if the code is not reentrant we will run into problems.

Let us also draw a distinction between thread safe code and reentrant code. A thread safe operation is one that can be performed from more than one thread at the same time. On the other hand, a reentrant operation can be invoked while the operation is already in progress, possibly from within the same thread. Or it can be re-started without affecting the outcome. See this code example from [Che04]:

```
int length = 0;
char *s = NULL;

// Note: Since strings end with a 0, if we want to
// add a 0, we encode it as "\0", and encode a
// backslash as "\\".

// WARNING! This code is buggy - do not use!

void AddToString(int ch)
{
    EnterCriticalSection(&someCriticalSection);
    // +1 for the character we're about to add
    // +1 for the null terminator
    char *newString = realloc(s, (length+1) * sizeof(char));
    if (newString) {
        if (ch == '\0' || ch == '\\') {
            AddToString('\\'); // escape prefix
        }
        newString[length++] = ch;
        newString[length] = '\0';
        s = newString;
    }
    LeaveCriticalSection(&someCriticalSection);
}
```

Is it thread safe? Sure - there is a critical section protected by the mutex `someCriticalSection`. But is it reentrant? Nope. The internal call to `AddToString` causes a problem because the attempt to use `realloc` will use a pointer to `s` that is no longer valid (because it got stomped by the earlier call to `realloc`).

Interestingly, functional programming languages (by which I do NOT mean procedural programming languages like C) such as Haskell and Scala and so on, lend themselves very nicely to being parallelized. Why? Because a purely functional program has no side effects and they are very easy to parallelize. If a function is impure, its functions signature will indicate so. Thus spake Joel¹:

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable. [Spo05]

This assumes of course that there is no data dependency between functions. Obviously, if we need a computation result, then we have to wait.

Object oriented programming kind of gives us some bad habits in this regard: we tend to make a lot of void methods. In functional programming these don't really make sense, because if it's purely functional, then there are some inputs and some outputs. If a function returns nothing, what does it do? For the most part it can only have side effects which we would generally prefer to avoid if we can, if the goal is to parallelize things.

¹"Thus Spake Zarathustra" is a book by Nietzsche, and this was not a spelling error.

Software Transactional Memory

Developers use software transactions by writing atomic blocks. These blocks are just like synchronized blocks, but with different semantics.

```
atomic {
    this.x = this.z + 4;
}
```

You're meant to think of database transactions, which I expect you to know about. The `atomic` construct means that either the code in the atomic block executes completely, or aborts/rolls back in the event of a conflict with another transaction (which triggers a retry later on).

Benefit. The big win from transactional memory is the simple programming model. It is far easier to program with transactions than with locks. Just stick everything in an atomic block and hope the compiler does the right thing with respect to optimizing the code.

Motivating Example. We'll illustrate STM with the usual bank account example².

```
transfer_funds(Account* sender, Account* receiver, double amount) {
    atomic {
        sender->funds -= amount;
        receiver->funds += amount;
    }
}
```

Using locks, we have two main options:

- Big Global Lock: Lock everything to do with modifying accounts. (This is slow; and you might forget to grab the lock).
- Use a different lock for every account. (Prone to deadlocks; may forget to grab the lock).

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

Drawbacks. As I understand it, three of the problems with transactions are as follows:

- I/O: Rollback is key. The problem with transactions and I/O is not really possible to rollback. (How do you rollback a write to the screen, or to the network?)
- Nested transactions: The concept of nesting transactions is easy to understand. The problem is: what do you do when you commit the inner transaction but abort the nested transaction? The clean transactional facade doesn't work anymore in the presence of nested transactions.
- Transaction size: Some transaction implementations (like all-hardware implementations) have size limits for their transactions.

Implementations. Transaction implementations are typically optimistic; they assume that the transaction is going to succeed, buffering the changes that they are carrying out, and rolling back the changes if necessary.

One way of implementing transactions is by using hardware support, especially the cache hardware. Briefly, you use the caches to store changes that haven't yet been committed. Hardware-only transaction implementations often have maximum-transaction-size limits, which are bad for programmability, and combining hardware and software approaches can help avoid that.

²Apparently, bank account transactions aren't actually atomic, but they still make a good example.

Implementation issues. Since atomic sections don't protect against data races, but just rollback to recover, a data race may still trigger problems in your program.

```
atomic {  
    x++;  
    y++;  
}
```

```
atomic {  
    if (x != y)  
        while (true) { }  
}
```

In this silly example, assume initially $x = y$. You may think the code will not go into an infinite loop, but it can.

References

- [Che04] Raymond Chen. The difference between thread-safety and re-entrancy, 2004. Online; accessed 8-December-2015. URL: <https://blogs.msdn.microsoft.com/oldnewthing/20040629-00/?p=38643/>.
- [Spo05] Joel Spolsky. The perils of javaschools, 2005. Online; accessed 8-December-2015. URL: <http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html>.