ECE459: Programming for Performance Winter 2023

Lecture 13 — Dependencies and Speculation

Patrick Lam and Jeff Zarnett *2022-09-04*

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 we don't have data races. 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. That is to say, there is a correct order for doing these two steps and getting the order wrong means we get the wrong outcome. If you want to bake a cake, you have to mix all the ingredients before you bake them; if you bake all ingredients and then mix them, whatever you get isn't a cake.

Remember, of course, that there are lots of things that don't have dependencies and can be done in arbitrary order (or even concurrently). If your household chores are to vacuum the floor and do the laundry, you get a valid outcome no matter what order you do them in, as long as you do both correctly. The outcome is the same!

There are two kinds of dependency that we'll cover and they are *loop-carried* and *memory-carried* dependencies.

Loop-Carried Dependencies. Let's start with the loop-carried version. In this kind of dependency, executing an iteration of the loop depends on the result of the previous iteration. Initially, `vec[0]` and `vec[1]` are 1. Can we run these lines in parallel?

```
let mut vec = vec![1; 32];
    /* */
vec[4] = vec[0] + 1;
vec[5] = vec[0] + 2;
```

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 i in 1 .. vec.len() {
    vec[i] = vec[i-1] + 1;
}
```

Nope! We can unroll the first two iterations:

```
vec[1] = vec[0] + 1;
vec[2] = vec[1] + 1;
```

Depending on the execution order, either `vec[2] = 3` or `vec[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*.

Okay, that's perhaps a silly example. Let's try a real problem. 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.
fn mandelbrot(x0: f64, y0: f64) -> i32 {
    let mut iterations = 0;
    let mut x = x0;
    let mut y = y0;
    let mut x_squared = x * x;
    let mut y_squared = y * y;
    while (x_squared + y_squared < 4f64) && (iterations < 1000) {
        y = 2f64 * x * y + y0;
        x = x_squared - y_squared + x0;
        x_squared = x * x;
        y_squared = y * y;
        iterations += 1;
    }
    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.

That doesn't mean that the problem cannot be parallelized at all, however. You can parallelize the Mandelbrot set calculation by computing the result simultaneously over many points at once, even if each point's calculation needs to be done sequentially. 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.

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

```
for i in 4 .. vec.len() {
    vec[i] = vec[i-4] + 1;
}
```

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

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

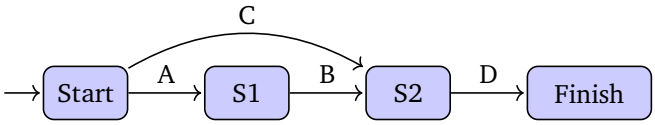
Memory-Carried Dependencies. On the other hand, a memory-carried dependency is one where the result of a computation *depends* on the order in which two memory accesses occur. For instance:

```
let mut acct: Account = Account {
    balance: 0.0f32
};
f(&mut acct);
g(&mut acct);

/* */

fn f (a: &mut Account) {
    a.balance += 50.0f32;
}
fn g (a: &mut Account) {
    a.balance *= 1.01f32;
}
```

What are the possible outcomes after executing `g()` and `f()` in parallel threads? Obviously, assuming they are properly wrapped in Atomic reference counted types to allow the code to compile.

Critical Paths

You should be familiar with the concept of a critical path from other courses (e.g., capstone design project) or just from project management you've experienced on coop terms. It is the minimum amount of time to complete the task, taking dependencies into account. 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.



Having a diagram like this makes it easy to see what can run in parallel and what steps need to happen in what order. You might not need such a thing, but they are super useful for explaining to nontechnical coworkers or senior management what is happening and how it's going.

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 will order, then this is a net savings of time on average.

You can even put some numbers on it! If successfully predicting your order saves 30 seconds per day, and if they're wrong, remaking it takes 3 minutes longer than normal, then if your order is unpredictable one out of every 10 days, how much time is saved? 9 days are correct for a savings of 4.5 minutes, from which we subtract the 3 minute penalty for being wrong, and we've saved 1.5 minutes in the 10 day period (or about 15 seconds per day

on average). Not huge, but imagine if the coffee shop can do that for every regular customer. If there are 100 regular customers that's 25 minutes of waiting saved per day! It all adds up.

Mind you, the idea of speculation isn't new in this course. 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:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let val = long_calculation(x, y);
    if val > threshold {
        return val + second_long_calculation(x, y);
    }
    return val;
}
```

Without more information, you don't know whether you'll have to execute `second_long_calculation()` or not; it depends on the return value of `long_calculation()`. Fortunately, the arguments to `second_long_calculation()` do not depend on `long_calculation()`, so we can call it at any point. Here's one way to speculatively thread the work:

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let t1 = thread::spawn(move || {
        return long_calculation(x, y);
    });
    let t2 = thread::spawn(move || {
        return second_long_calculation(x, y);
    });
    let val = t1.join().unwrap();
    let v2 = t2.join().unwrap();
    if val > threshold {
        return val + v2;
    }
    return val;
}
```

We now execute both of the calculations in parallel and return the same result as before. Is this the only way to do it? No. The current thread is a valid thread for doing work and we don't have to create two threads and join two threads: we can create one and maybe have less overhead.

```
fn do_work(x: i32, y: i32, threshold: i32) -> i32 {
    let t1 = thread::spawn(move || {
        return second_long_calculation(x, y);
    });
    let val = long_calculation(x, y);
    let v2 = t1.join().unwrap();
    if val > threshold {
        return val + v2;
    }
    return val;
}
```

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 `long_calculation`, the time T_2 that it takes to run `second_long_calculation`, 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:

```
fn do_other_work(x: i32, y: i32) -> i32 {
    let val = long_calculation(x, y);
    return second_long_calculation(val);
}
```

If the result of `value` is predictable, then we can speculatively execute `second_long_calculation` based on the predicted value. (Most values in programs are indeed predictable). If 90% of customers are using the standard billing plan, you might assume that that's correct in your calculations, and then change that later if it turns out you are wrong. Or you might have a software cache where you keep the latest state so you don't have to go to the database to check a value that changes rarely.

```
fn do_other_work(x: i32, y: i32, last_value: i32) -> i32 {
    let t = thread::spawn(move || {
        return second_long_calculation(last_value);
    });
    let val = long_calculation(x, y);
    let v2 = t.join().unwrap();
    if val == last_value {
        return v2;
    }
    return second_long_calculation(val);
}
```

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 `val != 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:

- `long_calculation` and `second_long_calculation` must not call each other.
- `second_long_calculation` must not depend on any values set or modified by `long_calculation`.
- The return value of `long_calculation` 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. As we've seen, Rust discourages those kinds of problems but doesn't forbid them: we can still have atomic types shared that would represent some sort of global state.

Software Transactional Memory

Somewhat related to the idea of speculation is the idea of software transactional memory. In this case, a group of changes are carried out on a speculative basis, assuming that they will succeed, and we'll check afterwards if everything is okay and retry if necessary [ST95].

There is a library for this in Rust[1] although we can't vouch for its quality or usefulness. Developers use software transactions by writing `atomic` blocks:

```
let x = atomically(|trans| {
    var.write(trans, 42)?; // Pass failure to parent.
    var.read(trans) // Return the value saved in var.
});
```

The idea resembles database transactions, which most likely you 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, and repeated retries if necessary to get it applied).

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 example[2].

```
struct Account {
    balance: TVar<f32>,
}

fn transfer_funds(sender: &mut Account, receiver: &mut Account, amount: f32) {
    atomically(|tx| {
        let sender_balance = sender.balance.read(tx)?;
        let receiver_balance = receiver.balance.read(tx)?;
        sender.balance.write(tx, sender_balance - amount)?;
        receiver.balance.write(tx, receiver_balance + amount)?;
        Ok(0)
    });
}
```

Using locks, we have two main options:

- Big Global Lock: Lock everything to do with modifying accounts. This is slow and serializes your program.
- Use a different lock for every account. Prone to deadlocks; high overhead.

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

[1] `https://github.com/Marthog/rust-stm`

[2] It turns out that bank account transactions aren't actually atomic, but they still make a good example.

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 façade doesn't work anymore in the presence of nested transactions. (The Rust library will panic at runtime if you try to nest 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.

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

```
fn what_could_go_wrong(x: TVar<i32>, y: TVar<i32>) {
    atomically(|t| {
        let old_x = x.read(t)?;
        let old_y = y.read(t)?;
        x.write(t, old_x + 1);
        y.write(t, old_y + 1);
        Ok(0)
    });
}
```

```
fn oh_no(x: TVar<i32>, y: TVar<i32>) {
    atomically(|transaction| {
        if x.read(transaction)? != y.read(transaction)? {
            loop { /* Cursed Thread */}
        }
        Ok(0)
    });
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can. That's because intermediate states can still become visible! Although the block is atomic, that just means the changes succeed or are rolled back as a group; it does not mean that another thread cannot read `x` and `y` and see them at some partial-completion state of the transaction. (Maybe this doesn't happen in the Rust implementation, but it certainly can in C/C++ versions of STM.)

References

[ST95] Nir Shavit and Dan Touitou. Software transactional memory, 1995. Online; accessed 1-March-2019. URL: `https://groups.csail.mit.edu/tds/papers/Shavit/ShavitTouitou-podc95.pdf`.