ECE459: Programming for Performance Winter 2021

Lecture 16 — Crossbeam and Rayon

Jeff Zarnett *2020-10-22*

Use Libs Wisely!

In previous courses, such as a concurrency course, there was a lot of expectation to do most things the hard way: write your own implementation and don't use libraries. In this course, such restrictions don't apply. In industry, you'll use libraries that have appropriate functionality, assuming the license for them is acceptable to your project. The two we'll talk about today, Crossbeam and Rayon are, for the record, Apache licensed, so it should pose no issue. In the previous version of the course where we used C and C++, we taught the OpenMP functionality, which is used to direct the compiler to parallelize things in a pretty concise way. The same idea applies here, except it's using the Crossbeam and Rayon crates rather than compiler directives.

Concurrency with Crossbeam

You'll recall from earlier when we introduced threads, that this doesn't work:

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

The problem was that the compiler can't tell for sure how long the data is going to live, and we said that we can get around this by moving the vector into the thread and returning it if needed again. Then later, when we wanted to share a Mutex between threads, we used the `Arc` type worked. We could have used the `Arc` type here too, but that's a bit of a pain if the thread is going to be short-lived. Crossbeam gives us the ability to create "scoped" threads. Scope is like a little container we are going to put our threads in. It allows the compiler to be convinced that a thread will be joined before leaving the scope (container). Alright, how about this?

```
fn main() {
    let v = vec![1, 2, 3];

    crossbeam::scope(|scope| {
        println!("Here's a vector: {:?}", v);
    }).unwrap();

    println!("Vector v is back: {:?}", v);
}
```

Wait a minute. That's not quite right, because if I add statements where we print the thread ID, I get this output:

```
main thread has id 4583173568
Here's a vector: [1, 2, 3]
Now in thread with id 4583173568
Vector v is back: [1, 2, 3]
```

All we did was make the container, but we didn't spawn any threads. Here's a better version:

```
fn main() {
    let v = vec![1, 2, 3];
    println!("main␣thread␣has␣id␣{}", thread_id::get());

    crossbeam::scope(|scope| {
        scope.spawn(|inner_scope| {
            println!("Here's␣a␣vector:␣{:?}", v);
            println!("Now␣in␣thread␣with␣id␣{}", thread_id::get());
        });
    }).unwrap();

    println!("Vector␣v␣is␣back:␣{:?}", v);
}
```

With output:

```
main thread has id 4439997888
Here's a vector: [1, 2, 3]
Now in thread with id 123145430474752
Vector v is back: [1, 2, 3]
```

There are still rules, of course, and you cannot borrow the vector mutably into two threads in the same scope. This does, however, reduce the amount of ceremony required for passing the data back and forth. Wrapping everything in `Arc` is tedious, to be sure.

Producer-Consumer with channels. One of the guidelines that Rust gives is that it's preferable to do message passing as compared to shared memory. Unfortunately, for something like the producer-consumer scenario, the standard multiple-producer-single-consumer channel we are provided won't do; we need a multiple-producer-multiple-consumer channel. And sure enough, Crossbeam gives us one.

Recall the multi-producer multi-consumer example from earlier. We had to define a shared buffer structure and use semaphores and a mutex to coordinate access, and we saw it's possible to get it wrong in when we drop the mutex. Here's the excerpts from the Crossbeam version. First, a little setup to create a bounded channel with capacity the same as the bounded buffer (100):

```
let (send_end, receive_end) = bounded(CHANNEL_CAPACITY);
let send_end = Arc::new(send_end);
let receive_end = Arc::new(receive_end);
```

This uses the bounded channel, which has a maximum capacity as specified. If it's full, the sender is blocked until space becomes available. You can also choose an unbounded channel, which allows an arbitrary number at a time (but of course, they have to go somewhere so uncollected messages still take up space in memory and memory is not infinite, but close...). And look how much simpler the consumer is:

```
    for _j in 0 .. NUM_THREADS {
        // create consumers
        let receive_end = receive_end.clone();
        threads.push(
            thread::spawn(move || {
                for _k in 0 .. ITEMS_PER_THREAD {
                    let to_consume = receive_end.recv().unwrap();
                    consume_item(to_consume);
                }
            })
        );
    }
```

Certainly that's a lot simpler and cleaner, but is it faster? Testing says yes! Hyperfine says the original producer-consumer-opt version takes 372 ms to run for 10000 items consumed per thread, and the version with the channel takes 232.

Try Again Later. Another small thing that Crossbeam enables is an exponential backoff. When attempting to access some resource, we acknowledge that it might not be available right now. If that's the case, an error is to necessarily fatal and the client might want to retry. However, it's very unhelpful to have a tight loop that simply retries as fast as possible. What you should do instead is an exponential backoff: wait a little bit and try again, and if the error occurs, next time wait a little longer.

The idea is that if the resource is not available, repeatedly retrying doesn't help. If it's down for maintenance, it could be quite a while before it's back and calling the endpoint 5 or 10 times every second doesn't make it come back faster and just wastes effort. Or, if the resource is overloaded right now, the reaction of requesting it more will make it even more overloaded and makes the problem worse! And the more failures have occurred, the longer the wait, which gives the service a chance to recover.

Eventually, though, you may have to conclude that there's no point in further retries. At that point you can block the thread or return an error, but setting a cap on the maximum retry attempts is reasonable.

The `Backoff` util from Crossbeam gives you this functionality. Each step of the backoff takes about double the amount of time of the previous, up to a certain maximum.

Here's an example from the Crossbeam docs of using the backoff in a lock-free loop. Here, the `spin()` function is used because we can try again immediately. We can do so because if the compare-and-swap operation failed, it's because another did the compare-and-swap and got a chance to run.

```
use crossbeam_utils::Backoff;
use std::sync::atomic::AtomicUsize;
use std::sync::atomic::Ordering::SeqCst;

fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
    let backoff = Backoff::new();
    loop {
        let val = a.load(SeqCst);
        if a.compare_and_swap(val, val.wrapping_mul(b), SeqCst) == val {
            return val;
        }
        backoff.spin();
    }
}
```

If what we actually need is to wait for another thread to take its turn before we go, we don't want to spin, we want to "snooze". This means we'll be waiting longer for the thread

```
fn spin_wait(ready: &AtomicBool) {
    let backoff = Backoff::new();
    while !ready.load(SeqCst) {
        backoff.snooze();
    }
}
```

In both cases, the `backoff` type has a function `is_completed` which returns true if the maximum backoff time has been reached and it's advised to give up. And an existing `backoff` can be re-used if it's reset with the unsurprisingly-named `reset` function.

Having read a little bit of the source code of the Crossbeam backoff, I'm not sure they implement a little randomness (called jitter). This is an improvement on the algorithm that prevents all threads or callers from retrying at the exact same time. Let me explain with an example: I once wrote a little program that tried synchronizing its threads via the database and had an exponential backoff if the thread in question did not successfully lock the item it wanted. I got a lot of warnings in the log about failing to lock, until I added a little randomness to the delay. It makes sense; if two threads fail at time X and they will both retry at time $X + 5$ then they will just fight over the same row. If one thread retries at $X + 9$ and another $X + 7$, they won't conflict.

The exponential backoff with jitter strategy is good for a scenario where you have lots of independent clients accessing the same resource. If you have one client accessing the resource lots of times, you might want something

else; something resembling TCP congestion control. See [Aeo19] for details.

Data Parallelism with Rayon

Looking back at the nbody-bins-parallel code that we discussed earlier, you may have noticed that it contains some includes of a library called Rayon. It's a data parallelism library that's intended to make your sequential computation into a parallel one. In an ideal world, perhaps you've designed your application from the ground up to be easily parallelizable, or use multiple threads from the beginning. That might not be the situation you encounter in practice; you may instead be faced with a program that starts out as serial and you want to parallelize some sections that are slow (or lend themselves well to being done in parallel, at least) without a full or major rewrite.

That's what I wanted to do with the nbody problem. I was able to identify the critical loop (it is, unsurprisingly, in `calculate_forces`). We have a vector of points, and if there are N points we can calculate the force on each one independently.

My initial approach looked at spawning threads and moving stuff into the thread. This eventually ran up against the problem of trying to borrow the `accelerations` vector as mutable more than once. I have all these points in a collection and I'm never operating on one of them from more than one thread, but a compile time analysis of the borrowing semantics is that the vector is going to more than one thread. The compiler can't prove to itself that my slices will never overlap so they can't all be mutable.

This is a super common operation, and I know the operation I want to do is correct and won't have race conditions because each element in the vector is being modified only by the one thread. I eventually learned that you can split slices but it was going to be a slightly painful process. You can use `split_at_mut()` and it divides the slice into two pieces... it's a start, but would require doing this multiple times and probably use recursion or similar. Further research eventually told me to stop reinventing the wheel and use a library for this. Thus, Rayon.

Parallelizing loops. Back on track. A quick glance over this program tells us it is likely that the slow step is computing the interactions. It's reasonably common that computationally-intensive parts of the program happen in a loop, so parallelizing loops is likely to be quite profitable in terms of speeding things up. This is something that Rayon specializes in: it's easy to apply some parallelization directives to the loop to get a large speedup at a small cost (here, cost is programmer time in writing the change and reviewing it).

The line in question where we apply the Rayon library is:

```
accelerations.par_iter_mut().enumerate().for_each(|(i, current_accel)| {
```

A lot happens in this one line, so we need to take a look at it. Normally, we iterate over a collection (here, the accelerations) using an iterator (and it's preferable to the `for` construction). That's normally a sequential iterator, and the convenient thing about Rayon is that we can drop it in pretty easily. We need to include the "prelude". The prelude brings into scope a bunch of traits that are needed for the parallel iterators. Then instead of iterating over the collection sequentially, we use a parallel iterator that produces mutable reference. We effectively ask to have the slices cut up into slices of size 1. I also ask for the enumerate option because I'm going to use the index `i`, and then I provide the operation that I want to perform: a for-each (where I specify the index name and the name of the variable I want to use for each element in the collection.

Doing the work. The description we just saw of dividing up the work is how work units are specified, but doesn't cover what actually happens on execution. After all, if we just divide up the work without having more workers to do it, we're not getting anywhere.

The Rayon FAQ fills in some details about how work gets done. By default, the same number of threads are spawned as available (logical) CPUs (that means hyptheading cores count). These are your workers and the work is balanced using a work-stealing technique. If threads run out of work to do, they steal work from other queues. The technique is based off some foundational work in the area, called Cilk (see [FLR98] for details).

Maybe too easy? We discussed already the effectiveness of this change. And here's how easy it was:

```
diff live-coding/L14/nbody-bins/src/main.rs live-coding/L14/nbody-bins-parallel/src/main.rs
2a3
> use rayon::prelude::*;
64c65
<     for i in 0..NUM_POINTS {
---
>     accelerations.par_iter_mut().enumerate().for_each(|(i, current_accel)| {
66d66
<         let current_accel: &mut Acceleration = accelerations.get_mut(i).unwrap();
79,80c79
<     }
<
---
>     });
```

Why does the ease of doing this matter? Every change we introduce has the possibility of introducing a nonzero number of bugs. If I were to rewrite a lot of code, there would be more opportunities for bugs to appear. This change is minimal and easier for a reviewer to verify that it's correct than a big rearchitecting. And it's faster – I can easily drop this in here and then move on to optimizing the next thing. Bugs slow you down.

It is important to note that the iterators do things in parallel, meaning the behaviour of the program can change a bit. If you are printing to the console or writing to a file or something, they can happen out of order when the loop is parallelized, just as they would if you wrote it so that different threads execute.

By that same token, the automatic parallelization doesn't prevent all race conditions. If you are trying to find the largest item in an array, we still have to use atomic types or a mutex or similar to ensure that the correct answer is returned. See a quick example below:

```rust
use rayon::prelude::*;
use rand::Rng;
use std::i64::MIN;
use std::i64::MAX;
use std::sync::atomic::{AtomicI64, Ordering};

const VEC_SIZE: usize = 10000000;

fn main() {
    let vec = init_vector();
    let max = AtomicI64::new(MIN);
    vec.par_iter().for_each(|n| {
        loop {
            let old = max.load(Ordering::SeqCst);
            if *n <= old {
                break;
            }
            let returned = max.compare_and_swap(old, *n, Ordering::SeqCst);
            if returned == old {
                println!("Swapped {} for {}.", n, old);
                break;
            }
        }
    });
    println!("Max value in the array is {}", max.load(Ordering::SeqCst));
    if max.load(Ordering::SeqCst) == MAX {
        println!("This is the max value for an i64.")
    }
}

fn init_vector() -> Vec<i64> {
    let mut rng = rand::thread_rng();
    let mut vec = Vec::new();
    for _i in 0 ..VEC_SIZE {
        vec.push(rng.gen::<i64>())
    }
    vec
}
```

Here, the vector is iterated over in parallel and it's not using the mutable parallel iterator. This code example uses an atomic type, but if I change this to use a `Mutex` instead of an atomic type (see code below) it increases the runtime from about 121.6 ms to about 871.7 ms (tested with hyperfine as per usual).

```
fn main() {
    let vec = init_vector();
    let max = Mutex::new(MIN);
    vec.par_iter().for_each(|n| {
        let mut m = max.lock().unwrap();
        if *n > *m {
            *m = *n;
        }
    });
    let m = max.lock().unwrap();
    println!("Max value in the array is {}", m);
    if *m == MAX {
        println!("This is the max value for an i64.")
    }
}
```

The non-parallel code takes about 136.2 ms. This problem doesn't have enough work to parallelize effectively. While the lock-free version speeds it up a small amount, the mutex version was easier to write (and is therefore what you would probably do) but made it much slower. So it's important to consider carefully when to apply the library, because a speedup is not guaranteed just by parallelizing a given loop.

References

[Aeo19] Aeoncase. Improve on exponential backoff, 2019. Online; accessed 2020-10-21. URL: `https://www.aeoncase.com/blog/posts/improve-on-exponential-backoff/`.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212223, May 1998. `doi:10.1145/277652.277725`.