ECE459: Programming for Performance Winter 2021

Lecture 16 — Rayon and Crossbeam

Jeff Zarnett *2020-10-16*

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.

As an aside as to why we're talking about this – 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. Rayon is, 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 Rayon crate rather than compiler directives.

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 hypthereading 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:

```
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.

Concurrency with Crossbeam

References

[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.