# Lecture 16 — Crossbeam and Rayon

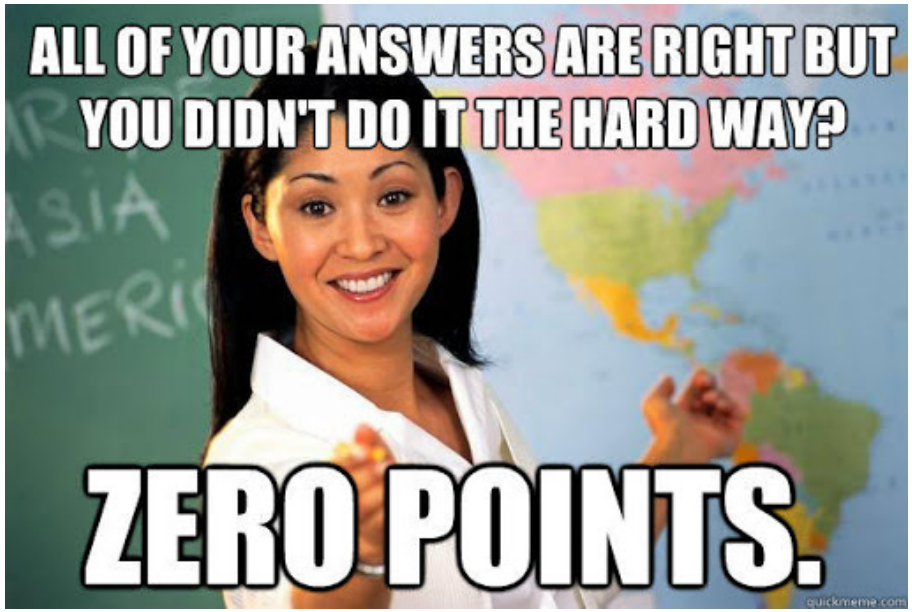
Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 25, 2020

# Doing it by hand...

In previous courses: do things the hard way: write your own!




In industry, you'll use libraries that have appropriate functionality.
... Assuming the license for them is acceptable to your project.

Goal: directed parallelization.

```
use std::thread;

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

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

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

Remember this? Still doesn't work: the lifetime of v is a problem.

For this situation, we used the `Arc` type sometimes.

Crossbeam gives us the ability to create “scoped” threads.

Scope is like a little container we are going to put our threads in.

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

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

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

Does this work?

# Did we forget something important?

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.

# Let's fix that...

```
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]
```

# Still not unlimited power...

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.

# Channels

Rust: use message passing!

Also Rust: doesn’t give you a multiple-producer-multiple-consumer channel.

Good news: Crossbeam gives us one.

# Produce, Consume, Repeat

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.

We saw it's possible to get it wrong in when we drop the mutex.

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

This is the bounded channel.

# Consumer Code: Simpler!

```
    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?

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 Later

Another small thing that Crossbeam enables is an exponential backoff.


Resources might not be available right now; retry later?

It’s unhelpful to have a tight loop that simply retries as fast as possible.

Wait a little bit and try again; if the error occurs, next time wait a little longer.

# Press F5 Until the Website Loads!

Repeatedly retrying doesn't help.

If it's down for maintenance, it could be quite a while before it's back.

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!

# It’s Hopeless

Eventually, 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.

# Backoff in a Lock-Free Loop

Backoff in a lock-free loop:

```rust
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();
    }
}
```

The `spin()` function is used because we can try again immediately.

# No, After You

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".

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

# We Give Up

In both cases, the `backoff` type has a function `is_completed` which returns true if the maximum backoff time has been reached.

An existing `backoff` can be re-used if it's reset with the unsurprisingly-named `reset` function.

I think they forgot something: Jitter!

# Jitterbug Into My Brain?


("OK Boomer...")

# Jitter

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, maybe you want something like TCP congestion control.

# Rayon

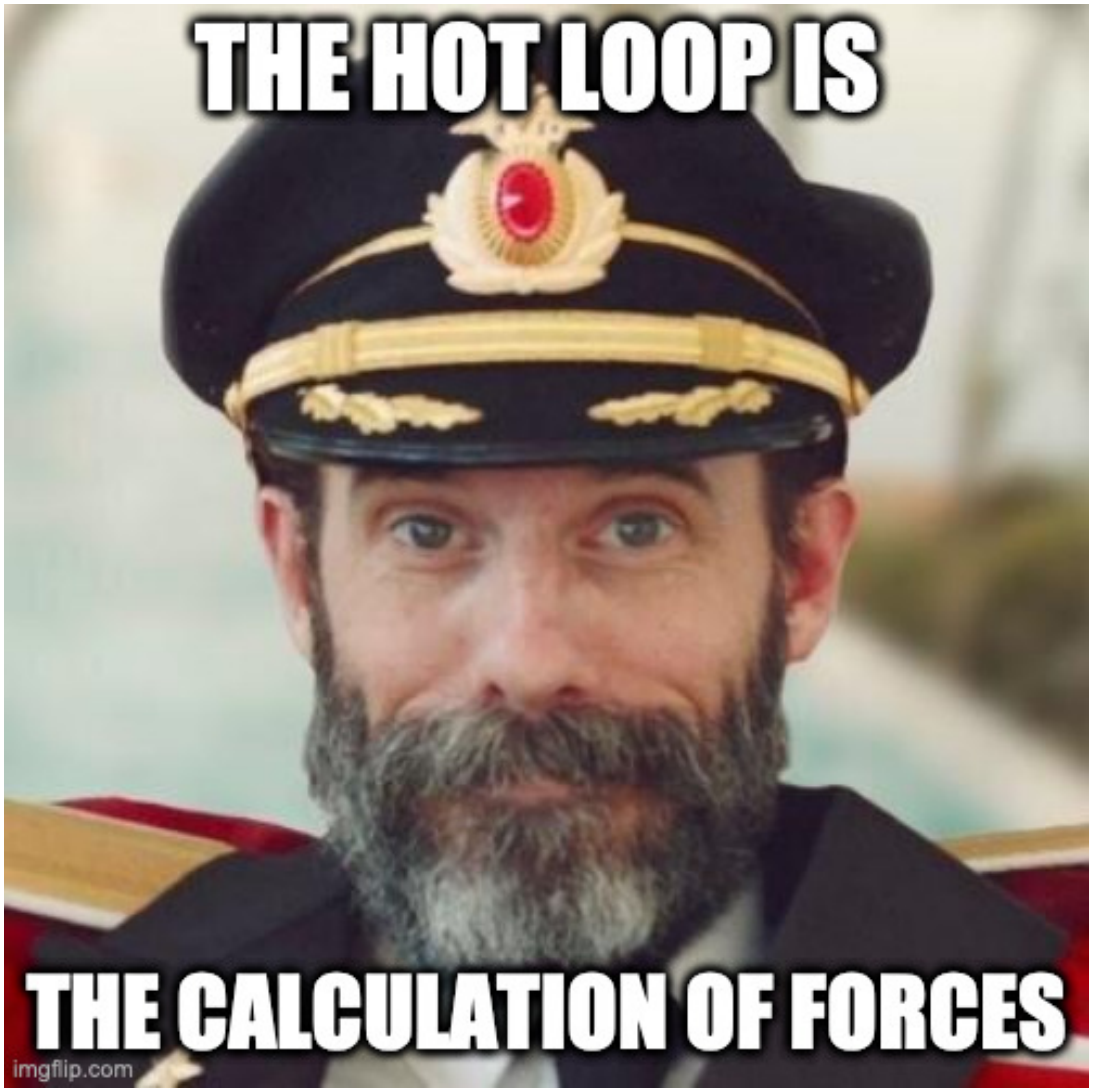
Let's revisit the nbody-bins-parallel program.

It uses Rayon: a data parallelism library!

In an ideal world, perhaps you've designed your application from the ground up to be easily parallelizable.

Rayon: parallelize some spots without a full/major rewrite.

# Thanks, Captain Obvious




We have a vector of points, and if there are *N* points we can calculate the force on each one independently.

# Goal: Parallelize

My initial approach looked at spawning threads and moving stuff into the thread.

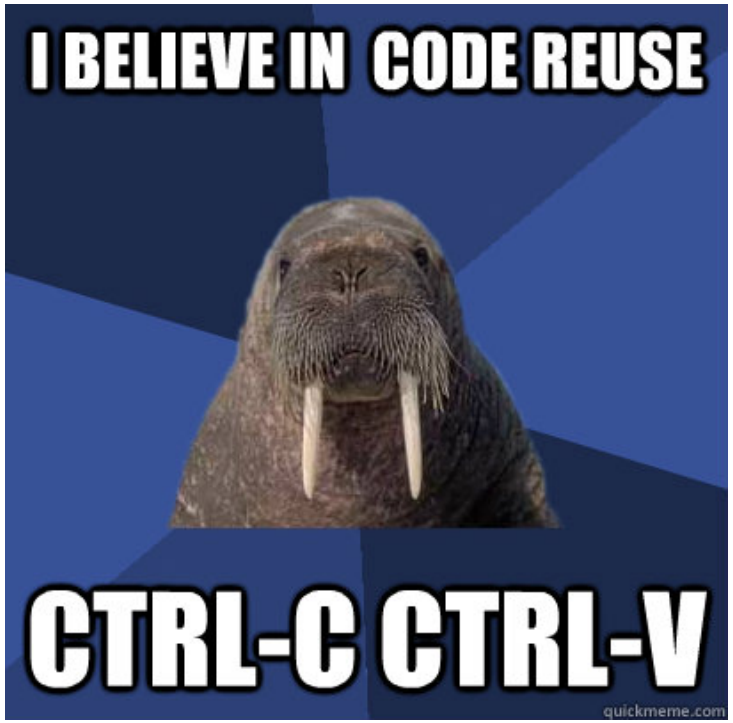
Problem: borrowing the `accelerations` vector as mutable more than once.

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.

# Stop Reinventing the Wheel

You can use `split_at_mut()` and it divides the slice into two pieces...

You have to do this a lot, and use recursion maybe?




Why not use a library?

# Apply Directly to Hot Loop

It’s reasonably common that computationally-intensive parts of the program happen in a loop, so parallelizing loops is likely to be quite profitable.

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 line...

# Demonstration

Let’s actually run the program and see the difference!

# Doing the Work

We covered how work-units are created, but not how work gets done.

We need more workers to get it done faster!

By default, the same number of threads are spawned as available (logical) CPUs.

Work is balanced using a work-stealing technique.

# Diff

Here's how easy the Rayon change 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?

# Ease of Use is Important

Every change has the possibility of introducing a nonzero number of bugs.

This change is minimal and easier for a reviewer to verify that it's correct than a big rearchitecting.

And it's less costly in dev time!

# Warnings About Parallelism

Iterators do things in parallel, meaning the behaviour of the program can change a bit.

Printing to the console, for example, could be out of order.

And you still have to satisfy the compiler there's no race conditions.

# Find the Max

Lets try to find the max of a randomly-filled array.

```
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 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
}
```

# Find the Max

```
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.")
    }
}
```

We use an atomic type to prevent data races.

# What About a Mutex?

```
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.")
    }
}
```

This increased the runtime from about 121.6 ms to about 871.7 ms.

The non-parallel code takes about 136.2 ms.

# Use Libs Wisely

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 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.