Week 6 Exercises: Sharing Data by Communicating

Yes we know, it's week 7, but these exercises pertain to material you learned in week 6, and who's keeping count anyway?

In this week's exercises, you'll get to appreciate the sleekness of channels, a concurrency abstraction you learned about last week.

The starter code is available on GitHub here.

Due date: Tuesday, May 26, 11:59pm (Pacific time)

Ping us on Slack if you are having difficulty with this assignment. We would love to help clarify any misunderstandings, and we want you to sleep!

Part 1: `parallel_map`

You want to share the joys of parallelism with your friends who haven't learned about synchronization primitives yet by implementing for them a special, speedy function. This function takes two arguments: a vector of type `Vec<T>` another function `f` which takes elements of type `T` as input and returns type `U` as output. It runs `f` on each input element in the input vector and collects the results in an output vector. Even better, it does this in parallel! The function looks something like this:

```
fn parallel_map<T, U, F>(mut input_vec: Vec<T>, num_threads: usize, f: F) -> Vec<
    where F: FnOnce(T) -> U  + Send + Copy + 'static,
          T: Send + 'static,
          U: Send + 'static + Default, {
    let mut output_vec: Vec<U> = Vec::with_capacity(input_vec.len());
    // TODO: in parallel, run f on each input element and collect the outputs,
    // in order, in output_vec
    output_vec
}
```

`Ok(reader)`, take a deep breath. There are a lot of trait shenanigans going on over here:

- You can read about `FnOnce` here. It's basically a trait that allows for us to pass in closures for `f` – it takes its inputs by value. It's basically a fancy typed function pointer. Rust also has other kinds of function pointer traits like `Fn` and `FnMut`. This isn't particularly important to understand for this assignment.
- Recall that the `Send` trait indicates that a type may be safely moved across thread boundaries.
- `Default` indicates a type that has some sort of default value implemented for it.
- The `'static` thing is a lifetime annotation – it says that any references that your object holds must have a static lifetime i.e. a lifetime as long as the running program. Here is a great discussion if you'd like to learn more. Here is the Rust Book on lifetimes.
- In summary, `parallel_map` takes in `input_vec` (as an owned type, so it can be consumed), `num_threads` (the number of threads that can execute in parallel), and a function `f` that takes as input values of type `T` and returns values of type `U`.

Your objective is to complete this implementation by using *channels as your only synchronization mechanism*. This might sound like a limitation, but trust me, this will make your life easier. You can implement a second version using mutexes and condition variables if you want to fully appreicate how nice it is to use channels.

In Lecture 12, Ryan showed how you can use channels to implement farm v3.0. Please make sure you understand that example before you embark on implementing `parallel_map`.

As is often the case with concurrency, your solution doesn't need to be very long (our solution is 43 lines of code long) but that doesn't mean it's trivial. You should carefully design your implementation before you code it up.

How you design `parallel_map` is completely up to you! You are free to use as many channels as you need and design the messages you send across those channels (of course you should strive for an implementation that is simple, correct, and efficient). As you're planning out your implementation, you should keep the following things in mind:

- The elements in the output vector need to positionally correspond to the elements of the input vector. That is: `output_vec[i] == f(input_vec[i])`. If you like, you may first implement a version that disregards order, then upgrade that to a version that respects the order.
- You are at the mercy of the thread scheduler. You have no idea what order threads will execute in, nor should you impose an order – doing so can severely limit the amount of parallelism you achieve by making your code unnecessarily sequential.
- You can send any type of value through a channel that you like, including structs or tuples. (Side note: the values need to be `Send` in order to be sent through a channel, but that shouldn't be a problem for you.)
- You can assume `f` won't have any side effects i.e. it will not mutate any external state in your program.
- Channels, like pipes, are unidirectional.

- You need to `drop` the sender for receivers to know that there's nothing more to receive (you can see this in the farm example). If your code is hanging, it's probably because you didn't properly drop a sender.
- You need to move things out of your input vector in order to properly transfer ownership. (Think `pop`.)
- Your implementation should only spawn a total of `num_threads` threads. (If you're familiar with the CS 110 ThreadPool, this is basically spinning up a short-lived ThreadPool to execute `f` over `input_vec`, aggregating the results. Don't worry; because of the beauty of channels, this implementation will not be as complex as ThreadPool.)
- Strive for efficiency – don't remove from the front of the vector, that is a O(n) operation that could have been O(1).
- Strive for efficiency – you know exactly how many things to expect in your output vector. Try not to do any unnecessary sorting. That is a O(n log n) operation that didn't need to happen.

(Optional) Feel free to do something fun with the `parallel_map` implementation – use it to revamp the link explorer lecture example. Use it to implement a parallelized Mandelbrot Set generator. It's a very versatile function – the possibilities are endless!

Part 2: Weekly Survey

Please let us know how you're doing using this survey.

When you have submitted the survey, you should see a password. Put this code in `survey.txt` before submitting.

Optional Extensions

The `parallel_map` function you implemented effectively spins up a ThreadPool, uses it to execute the maps, and then destroys the ThreadPool. Implement a proper ThreadPool that only destroys its worker threads when `drop`ped and give it a `parallel_map` function as well that accomplishes what you did in Part 1.

If you thought `parallel_map` was fun, wait till you hear about `paralle_reduce`. Suppose you have some commutative aggregation function – say `+` for example. If you wanted to add up the numbers in a `Vec` of size 8, you could do it the boring way – by taking a linear pass through the vector and accumulating the sum. Or you could do something like this:

```
 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8
 \   /   \   /   \   /   \   /
   3       7       11      15
```

where the sums (1 + 2), (3 + 4), (5 + 6), (7 + 8) are all done in parallel, then you do another round of parallel sums – this time (3 + 7) and (11 + 15). And then you do one final sum to get your result. This is precisely what a `parallel_reduce` implementation should do. This presents some new synchronization challenges. Although your `parallel_map` implementation should serve as a good starting point. Better yet, you can tack this `parallel_reduce` function onto your ThreadPool implementation, and now you've got yourself a really fancy ThreadPool.

Some CS161 food for thought – what would the asymptotic runtime of `parallel_map` be if you had infinite parallelism and each binary operation was O(1)? What if you could run M threads at once? What if each binary operation took time proprotional to the number of elements it aggregated over? What if each binary operation took time that varied according to a geometric distribution with success probability... jk haha.