Parallelized Prefix Sum through Passing Accumulated Values Between Concurrent Serial Prefix Sums

Alexander Pietrick
[redacted]

March 1st, 2022

The Basic Idea

At its highest level, my algorithm breaks the input array into a set of chunks for serial prefix sums to be performed. These chunks have metadata (chunk headers) that store information about them such as their location, current accumulated value, and whether they have finished calculating their prefix sum or not.

Once a chunk has completed its prefix sum, that chunk will find where the next unfinished chunk to its right is currently at in its serial prefix sum. The completed chunk will increase the accumulated sum of the incomplete chunk by its own accumulated sum (which is the last number of its prefix sum). Then the complete chunk will increase the numbers between where its prefix sum ends and the current index of the unfinished chunk it just modified by its accumulated sum.

Each thread is assigned a chunk to handle. All of these chunks/threads work independently of each other, which allows my algorithm to take advantage of concurrency. The only time where my algorithm needs there to be absolute coordination between chunks is when a chunk is finished and needs to starts to look at the locations of other chunks. This was the major hurdle for me to overcome in my implementation.

Background

The idea for this algorithm came to me when I was discussing the faults of another student's algorithm. Specifically, [redacted] and I were discussing [redacted]'s first solution (a parallelized prefix sum approach with sequential dependencies). We had overheard [redacted] discussing it with [redacted] and were curious as to why it was running so fast (~1.09X).

As we were talking about how [redacted] 's algorithm worked, I made the observation that it spends quite a lot of time waiting for the previous sections to finish. With [redacted]'s first algorithm, a section needed to wait for all of the sections before it to complete before even starting on its own prefix sum. This got me thinking about how I could design an algorithm that would avoid this, allowing for sub prefix sums to calculate in any order and still take advance of dividing the task up into smaller problems.

The idea of having a completed section update the section to its right's current accumulated value and then fix the indices whose prefix sums were calculated before this update (what I later called the missed indices) naturally came to mind. The larger task for me was figuring out how to ensure that threads weren't colliding with each other and coordinating the crucial functions that this algorithm would require. This was my first experience with multi-threading, so it was a bit intimidating, but I think I have designed an algorithm that successfully ensures thread-safe, the correctness of the output, and performance that is relatively similar to a serial solution.

Algorithm Design

While I feel that my approach is easy to understand once you see how it tackles the sub prefix sums tasks, there are quite a lot of complicated moving parts happening all at once. Luckily, most of them don't collide with each other, so I can go through them

independently. Building off of what I explained in **The Basic Idea**, consider the following pseudo-code:

Read the input file and get the input array
Calculate the size of a chunk using length of file and number of threads allowed
Create all needed chunk header structures and initial
Create threads that run a serial prefix-sum task on all chunks

Wait for all threads to stop
Then print out the input array

For each chunk header:
    While chunk hasn't finished calculating prefix sum:
        If chunk isn't supposed to wait, continue calculating the prefix sum
        If chunk is supposed to wait, wait for the signal to start again

    While chunk hasn't added its sum value and there are unchecked chunks:
        Tell next chunk to wait and get its lock
        If next chunk hasn't added its prefix sum yet:
            Increase next chunk sum value with chunk's own
        If next chunk has added its prefix sum:
            set next chunk to the chunk after next chunk
        Unlock next chunk and signal it to continue

    If chunk was able to add its prefix sum
        Start increases values by prefix sum until next chunk's current index

Increase the value number at the given index by the given amount
If more numbers need to be increased and there is another free thread available

Split task in two and start filling the gap with the other thread

If no more numbers need to be increased

Terminate this thread

I promise you this isn't the writings of a mad man. The above pseudo-code showcases the general logical flow of the sub-parts of my algorithm: I will be going into greater depth below for each part.

If you take anything away from this pseudo-code, look at how it's structured into three general tasks: setup (A Threading), serial prefix sum and transition (B Threading), and then increasing missed indices (C Threading). Each of these tasks is done by a different number of threads and signifies a different step in process of my algorithm. Working through one of these tasks is what I call "Threading" and there are letters assigned to each. Other than A Threading (which is the setup task), these tasks run simultaneously, with only B Threading (serial prefix sum and transition) needing access to other threads' information.

Before I get to that, Let me first explain A Threading.

A Threading: Setup

A Threading is associated with the red text in the pseudo-code and is the easiest to understand. It is only done once by the main thread (there is only one A Thread) and simply just prepares the program to start the body of my algorithm.

Using the read_input_vector function that was given to us in scan_serial, the A Thread will read the input file into an array.

Next, the A Thread prepares to create all the needed chunk headers for B Threading. This is done by first figuring out how many B Threads (threads that are created specifically for B Threading) the program can even create and how much of the array a single B Thread will be in charge of. The A Thread tries to use as many B Threads

as it can, only having leftover threads if the number of threads available is larger than the size of the input array. The size of the B Threads can vary from 1 (where each B Thread is only in charge of 1 item in the array) to the size of the input array (if there is only one available thread).

After creating the chunk headers and placing them into a global array, the A Thread then creates B Threads (using pthreads) and tells them to start working (B Threading) with their respective chunk header. B Thread 1 works with chunk header 1, B Thread 2 works with chunk header 2, etc. Along with creating B Threads, the A Thread also creates a lock for each B Thread and store it in a different array but with the same index as the chunk header.

Finally, the A Thread will stop and wait to be signaled. Specifically, it is waiting for the number of available threads to equal the total number of allowed threads, which only happens when all the B and C Threads have finished. Once it receives the signal, the A Thread will then print out the input array and terminate.

The A Thread will always be the first and last thread to run. It is the branching point for the rest of the threads and my algorithm. As for how it interacts with the other threads, it really doesn't much. Most of the time, the A Thread is waiting for its signal to wake up and then print out the input array. Stepping through the algorithm, next is B Threading.

B Threading: Serial Prefix Sum and Transition

B Threading is the bulk of the complexity and work of my algorithm. It is represented by the blue pseudo-code. There are two tasks that B Threading does: calculate a chunk's prefix sum through a serial solution and then transiting into a C Thread. While these two tasks aren't super complicated separate, a B Thread needs to be prepared to be interrupted for my algorithm to work.

B Thread Chunk Header

B Threading needs a unique struct in order to work called a B Thread chunk header (or simply just chunk header). This struct holds metadata about the B Thread. The exact code for the struct is below.

Importantly there are a few variables are that absolutely needed both for the B Thread to do all its work (both the serial prefix sum and transition task) and for ensuring correctness of the end result. Some of these fields are self-explanatory (current_index, current_sum, starting_index, ending_index, and chunk_id), but I would like to quickly define the others.

```
struct B_thread_chunk_header {
    bool finished;
    bool added_sum;

    bool wait_for_other_chunk;
    pthread_cond_t start_summing_again;

    int current_index;
    int current_sum;

    int starting_index;
    int ending_index;

    int chunk_id;
};
```

The boolean finished is whether the B Thread has finished calculating the chunk's prefix sum yet. The booleans added_sum and wait_for_other_chunk and the pthread_cond_t start_summing_again relate to interrupting other B Threads to add your own prefix sum to their current value.

Serial Prefix Sum

Let me first explain the easy part of B Threading, calculating a chunk's prefix sum, which is fairly straightforward. The B Thread just goes from its starting index to its ending index and conducts a serial prefix sum. This value is stored in the chunk header with each step.

However, a B Thread can only work on its prefix sum if it isn't supposed to be waiting for another chunk (wait_for_other_chunk = false). If that is true, the B Thread must wait for a signal (start_summing_again) to be sent in order to start working again. A B Thread needs to behave like this in order to not run into concurrency errors when another B Thread is trying to transition.

Once a B Thread is finished with its own prefix sum, it sets its finished boolean to true and then moves on to the transition step of B Threading.

Transition

The transition step of B Threading is one of the most important and complicated sections in my algorithm. What needs to happen is a B Thread needs to find a chunk with a chunk id larger than its own (i.e a chunk that is located further to the right in the array) and hasn't added its prefix sum to another chunk yet. Once it finds a chunk, the B Thread should add its completed accumulated value from its prefix sum to that of other B Thread.

What makes this step challenging is coordinating the chunks so that they are ready to be accessed and changed by another B Thread that wasn't assigned to them. The B Thread which has the incompleted prefix sum chunk needs to know that it has to give up its lock and then wait to continue again. I achieve this by having the B Thread working in the transition step change a variable in the chunk it's looking at

(wait_for_other_chunk). This B Thread then waits to grab the other B Thread's chunk header's lock.

Going back to the serial prefix sum step of B Threading, the other B Thread will see that this boolean is switched to true and will release its lock on the chunk header and wait to get it back again. **This needs to happen when the B Thread with the incomplete prefix sum chunk is ready to give it up. If not, there is the chance of having a mismatch between a chunk header's current index and current value, which would produce the wrong answer.**

After the B Thread in the transition step has the lock for the other chunk header, it checks to see if that chunk has added its sum yet. There will always be a chunk header to the right of the current chunk that hasn't added its sum yet (the last chunk). It is preferred that this chunk isn't far away from the current chunk due to computations in the next step.

Assuming that the chunk the B Thread is looking at has not added its sum yet, it adds its accumulated prefix sum to the grabbed chunk header and also grabs where that chunk header is located in the array. Releasing the lock so the other B Thread can work again, the B Thread is through its transition step and set its added_sum to true.

All that is left is increasing the indices which are supposed to have the prefix sum but currently don't. C Threading handles this.

C Threading: Increase

C Threading is supposed to make up for the missed indices which never received a chunk header's serial prefix sum when they were supposed to. What happens is that a C Thread is created (either by a B Thread becoming one or by creating a new thread) and it is passed its current index, ending index, and then the value which should increase every value by in between those indices.

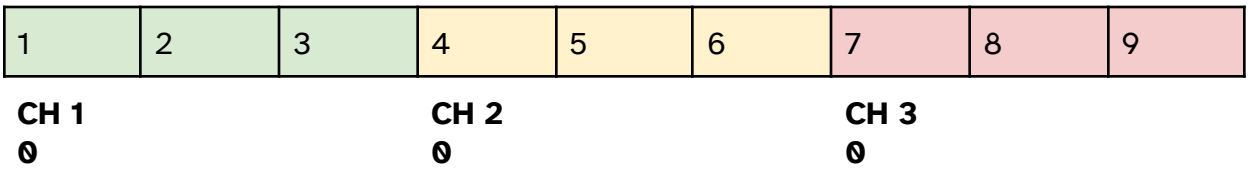
A C Thread will just iterator through the array and change these values. If another thread is available to use, a new C Thread will be created and the work will be split in half.

C Threads does not need to use locks as the operation they are performing is atomic and they never care what the actual value of the index is before increasing.

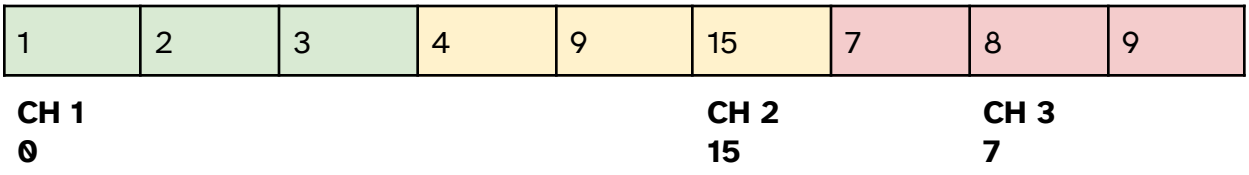
Once a C Thread has completed its desired work, it will terminate. If terminating the C Thread will increase the number of threads available to the number of allowed threads, it will first signal the A Thread to wake up and try to see if it can print out the input array.

Step-Through Example

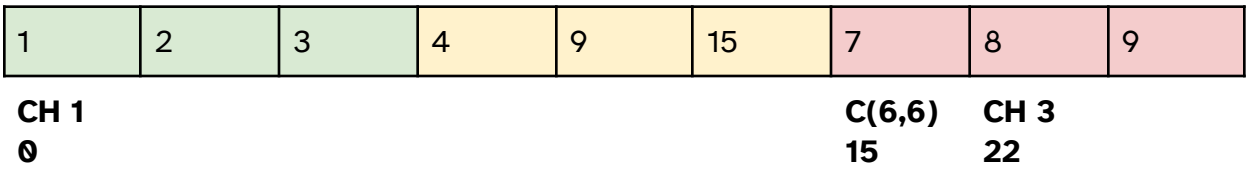
Now I'm going to step through an example of how my algorithm works. Consider an input of size 9 which allows for 3 threads to work. Here is what the input array would look like after A Threading:



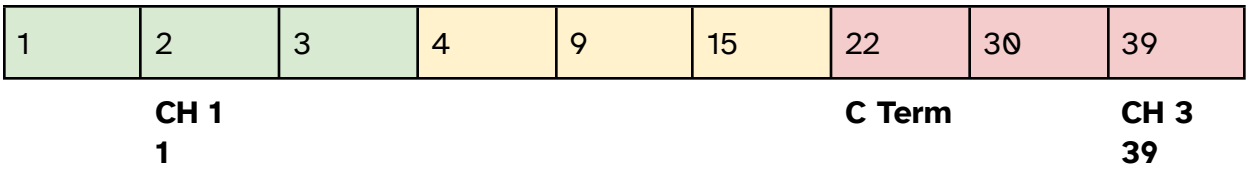
There are 3 B Threads each of length 3; they are all colored differently. B Threading hasn't started, so all of their chunk headers are at the first index in their associated chunk and their accumulated values are 0. Now the program is going to let all of the B Threads start working.



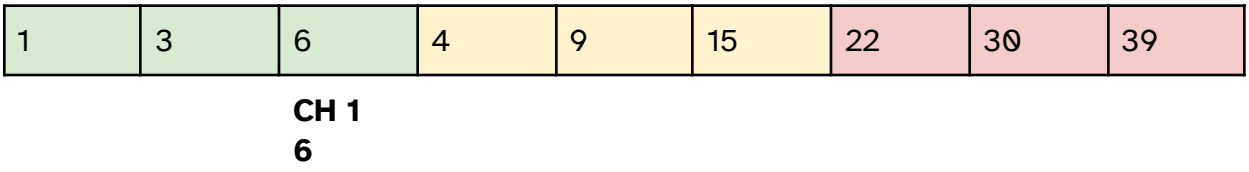
Chunk 2 has reached the end of its prefix sum and has found its accumulated value to be 15. Chunk 2 now looks to see where it can add its accumulated value and transition to becoming a C Thread.



Since chunk 3 hadn't finished yet, chunk 2 was able to add its own accumulated value to it. Now chunk 3 has an accumulated value of 22 (= 15 + 7) and chunk 2's B Thead has become a C Thread from index 6 to 6 with an increase by of 15.

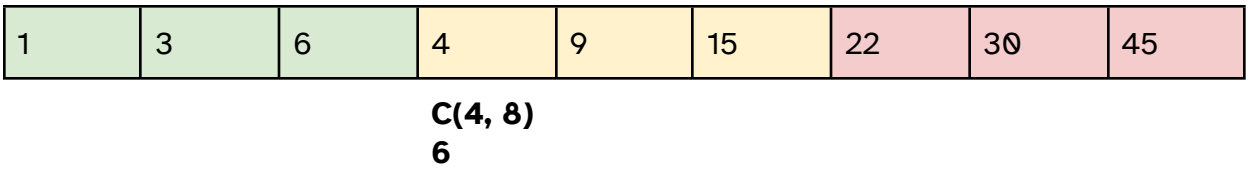


A few things have happened here. First, the C Thread that was updating index 6 reached the end of its task. This means that the thread terminated and now we have one extra thread available. Second, chunk 3 is completed its prefix sum. However, since there are no chunks after chunk 3, chunk 3 now terminates without setting its added_sum to true. This is important for the B Thread in charge of chunk 1. Let's continue.

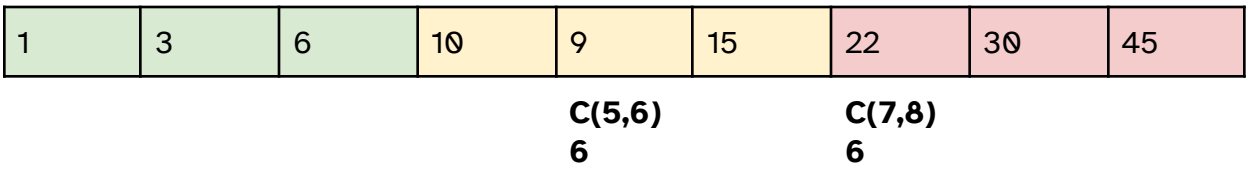


Finally, the B Thread in charge of chunk 1 has finished. It tries to add its accumulated value to the chunk 2 header; however, chunk 2 has already added its prefix

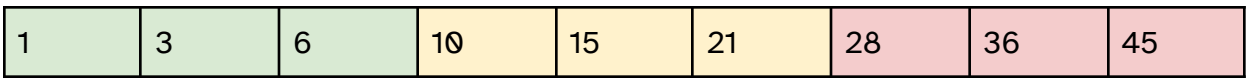
sum. So now chunk 1 adds its accumulated value to chunk 3's header (which hasn't added its sum since it's the last thread). Now the input array looks like this:



The B Thread from chunk 1 has become a C Thread from index 4 to 8 and is increasing the value by 6. Let's do one iteration of C Threading and see what happens.



The C thread adds the increase by and since there is now at least one more thread available, it creates another thread and splits its work in half. Let's finish these last two iterations and get the final prefix sum:



Now we have our completed prefix sum. Note that how the algorithm actually runs heavily depends on the order that the scheduler schedules the B Threads. There are some orders that are more desirable for this algorithm. If chunk 1 would have completed its prefix sum before chunk 2 was finished, then no C threading would have needed to be done from that transition.

The best order of completing prefix sum here would be chunk 1 before anything else, chunk 2 before anything else, chunk 3. There would be no C Threading needed if this was the case and the prefix sum would be calculated sequential here.

Of course, the opposite is true. The worse possible ordering here would be: chunk 3 before anything, chunk 2 before anything else, and then chunk 1. This would have produced the maximum amount of work for C Threading of an input array of this size.

Regardless of the order in which the scheduler has the chunks finished, my algorithm will ensure that the final answer is correct.

Coordinating Threads and Avoiding Deadlocks

Coordinating the A Thread and B Threads

There are two locations in my algorithm where I need to coordinate threads: getting the A Thread to wait until all other threads have terminated and stopping a B Thread from working on its prefix sum until another B Thread has completed work on its chunk header. In both of these cases, I use a combination of locks, signals, and variables.

For getting the A Thread to wait, I put the thread in a loop while the number of available threads is not equal to the total number of threads allowed. In this loop, there is a cond wait function where the A Thread will wait for a signal to be given to it to try to escape the while loop. This signal is given when a thread terminates and notices that it is the last thread (other than the A Thread). If the A Thread wakes up and isn't supposed to print yet, it will just loop and wait again. There will never be a problem with the A Thread waiting on a signal that will never come as all B/C Threads are terminated with a specific function, which would send that signal if the condition is met.

Coordinating the B Threads is done in a similar way with one extra step. The B Thread that needs another B Thread to halt needs to first change a boolean in that B Thread's chunk header. This boolean tells the B Thread currently working on its prefix sum to pause and give up its lock when it gets to a good spot. The B Thread that just gave up its lock will then wait for a signal to continue again.

Now the B Thread with the completed prefix sum has the other's B Thread's lock; however, now it must get its own again. It has to release its lock when waiting for the

other B Thread's as a chunk below it could have finished and wanted to transition to. It's better for the lower lock to transition first to reduce C Threading; therefore, the program will give up the B Thread's lock while waiting on another B Thread's lock.

Avoiding B Threads Deadlocking

If two locks are needed for a B Thread's transition step, then how is the program not end up in deadlock? The program will not end up deadlocking as two of the four deadlocking conditions are not met.

For a B Thread to get another B Thread's chunk header lock, there is resource preemption. The boolean that the B Thread changes will signify that the other B Thread should prepare to give up its lock. Without such preemption, the B Thread won't give up its lock until after completing its prefix sum, which isn't preferred in this algorithm. B Threading transition takes priority over everything else: it's specifically the step in this algorithm that is supposed to reduce the amount of wait/run time. It is crucial that it's both fast and safe for the threads. Preempting the B Thread's chunk header allows for B Thread to quickly wrap up its work (making it safe) and give up its lock for some time (fast).

As for the B Thread now needing to wait to get its own lock again, it is critical that there are never any circular waits. A B Thread never cares about the lock below it (chunk with id less than its own). Such the number of chunks is finite, there will always be a finite number of B Threads possibility holding its own lock.

Consider an example of 3 chunks: chunk 3 is still doing its prefix sum but is waiting. Chunk 2 has chunk 3's lock, but Chunk 1 has chunk 2's lock. The algorithm will always be able to get out of this by using the fact that no chunk other than 1 needs to access chunk 1's lock. Chunk 1 will then be able to get its own lock again and then free chunk 2's lock.

This is always true for when a B Thread needs another B Thread's lock. Going back to the example, let's say chunk 3 finished before chunk 2 and chunk 1 finished. Now chunk 2 needs to get chunk 3's lock. Chunk 2 will be able to get this lock with ease as

chunk 3 has transitioned and there are no other chunks above chunk 3 that could the lock.

While there might be the case of unfortunate timing that causes long wait lines, my algorithm will never deadlock due to the inherent nature of finitely and preempting resources.

Maximizing Opportunities for Concurrency and Performance

Concurrency and Performance while B Threading

The whole idea of B Threading is to take advantage of the fact that you don't need every number before an index to start getting prefixes. By splitting up the array into X amount of chunks, you are able to get the final prefix for 1 chunk and a portion of the final prefix for X-1 chunks with just one iteration and not passing values.

B Threading is great for concurrency as most of the time, a thread only needs the chunk header associated with itself. Assuming the chunk are of size N, a chunk header only needs a lock of another chunk 1 time at location N. Meaning that N-1 times, their work is self-contained.

Furthermore, running the B Threads concurrently allows for the opportunity to reduce later work. Think back to [redacted]'s parallelized prefix sum approach with sequential dependencies. Each section had to wait for all the other sections before it to finish before even starting to do a little work. My algorithm allows for chunks to work whenever and not have to continuously wait for other chunks.

Consider the opposite approach now, what if I had each chunk do its prefix sum but wait to pass on its accumulated value until all chunks had finished (if I used a barrier)? There would be a point where all but one chunk would have finished, but they would still have to wait for that chunk. Like I stated earlier, B Thread's transition step is the most crucial part of my algorithm as it allows all the other chunk freedom to complete their prefix sums in any order.

Regarding performance, as stated earlier, each chunk is self-contained and its performance isn't really dependent on other chunks. However, while that is true, I still had to make sure that a chunk had its associated lock when generating its prefix sum.

At first, I had the B Thread lock at the start of the while loop and then unlock at the end, allowing for other B Thread in the transition step to grab the lock in-between. This worked fine; however, it wasn't optimal as there was wasted time in locking and unlocking for the times where another B Thread didn't need the lock.

So, I changed my B Thread to work under the assumption that it always had the lock and that if another B Thread needed the lock, it would preempt it. This optimization increase the performance of my algorithm by ~0.15X and led to a much more efficient locking and signal relation between B Threading's prefix sum and transition steps.

In all, B Threading is written to maximize the number of B Threads running it without having to wait and to increase the greater performance of my algorithm.

Concurrency and Performance while C Threading

C Threading was originally designed to take care of very small repetitive tasks that didn't require any condition checking. Increasing missed indices is a task that needs to get done before the A Thread can print, but doesn't need to require locks for checking the values at all.

Because of when C Threading happens in relation to B Threading, it is a fact that C Threading will always happen over an area that has already had its value included in a serial prefix sum and will never be read (during the algorithm's work cycle) again. Since addition is an atomic function, C Threads do not need to fiddle with locks for trying to safely get and increase values: they just simply need to get the work done.

Furthermore, C Threading has increased performance and concurrency when there are additional threads available to divide some of a C Thread's work between. The quicker the calls are made to increase these values, the quicker that the program is able to finish. A C Thread doesn't care about other C Threads; they don't matter at all to their

task at hand. C Threads are simply just machines that take a number as an input and output the same number but increased by a set amount.

Conclusion

Overall, I am quite proud of my algorithm and how I was able to implement it for this project. I believe it averages ~1.05X and I have seen it get below 1X a couple of times. There are definitely some more optimizations that I could make in order to improve my time. I am pretty sure I could fiddle around with how I'm implementing C Threads a bit more, but I am more than content with the project for now.

Special Thanks To

- [redacted], who indirectly helped me create my algorithm by leading me to the concurrent prefix sums structure and got me thinking about how these prefix sums interact with each other
- [redacted], who helped me with some concurrency debugging (primarily with waiting on condition variables) and was a driving force in my motivation in improving my time, as I wanted to beat his time (Although, I believe his algorithm is slightly faster than mine still)
- [redacted] and [redacted] for both checking my algorithm to make sure that there weren't any sequential dependencies (giving me the permission to implement for this project) and helping me with understanding and debugging some of my concurrency issues
- The grader who had to read this all. I'm really sorry; it honestly got a bit out of hand...