CMSE 401 Project

Matthew Roberts

Summary:

The purpose of CMSE 401 was to learn different methods and techniques to break up, parallelize, and speed up code, primarily in C++. We utilized OpenMP, MPI, and “manual” parallelization methods to do our best to make the code faster, no matter its contents. So naturally for my final project I left everything that we had learned in class behind and embarked on a quest to learn a new programming language. Okay not really. Let’s take a look at what actually happened.

The code:

Dr. Appelö and his team provided me with a study that they worked on and was published last year. The code within was primarily made up of julia code, something I was not familiar with. The study is called “Universal AMG Accelerated Embedded Boundary Method Without Small Cell Stiffness”, and its purpose is to solve some variation on the wave equation.

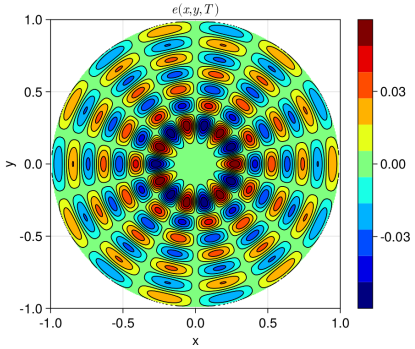
The process:

The first step that I attempted to take was to understand the math. I watched several youtube videos but was unable to fully understand it (or even remotely). One of the graduate students did their best to teach me what was going on, but I was also unable to fully grasp the mathematical concepts that were fundamental to building the code. I moved on.

Fortunately modifying code doesn’t require you to understand the math behind it. It’s just a bonus. I would still be able to finish this project so long as I followed two guiding goals.

1. Ensure results look the same every time you run.

A series of plots were produced when running the code, as well as a results.m file that showed the answers to various calculations. If I could modify the code and still get the same plots and numbers as before, It would mean that I hadn’t broken the code!



The figures above show the plot and answers to calculations that were to be kept the same. Any deviation in them would imply that I had unknowingly modified a calculator I shouldn't have.

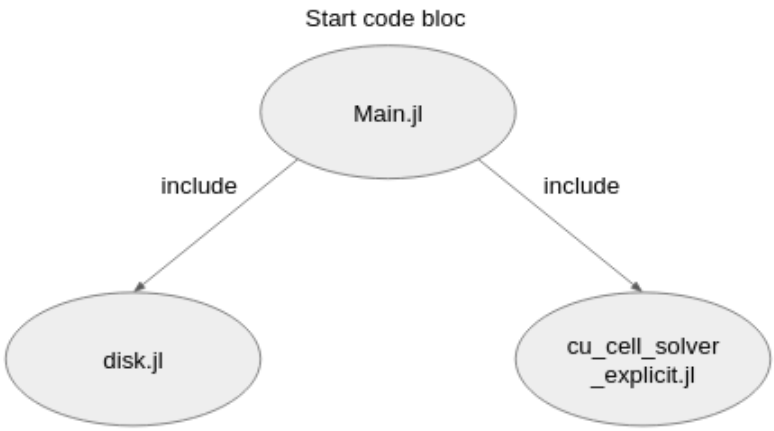
2. Actually speed up the code

This was the point of the project. Although it should go without saying, this was the second guiding goal of the project to get me by.

With the two goals established I had to fully understand how I could parallelize in Julia similar to what we had done in class. Luckily for me, Julia has its own built-in multi-threading syntax that was almost as easy, if not easier than OpenMP. The syntax looked similar to the figure below.

```
julia> Threads.@threads for i = 1:10
           a[i] = Threads.threadid()
       end
```

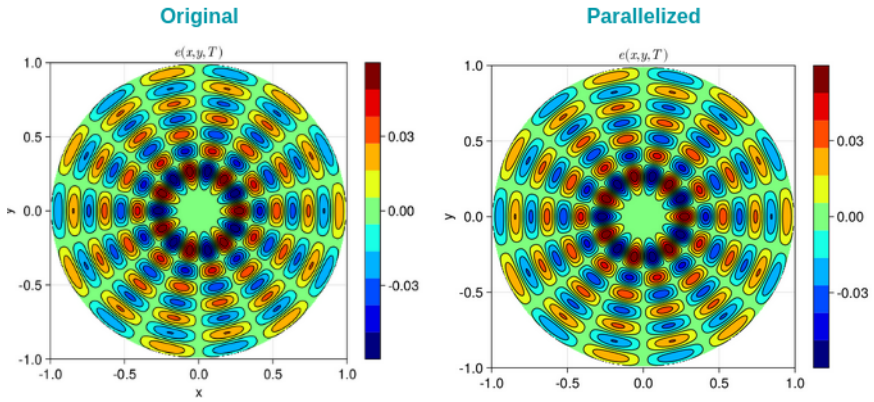
All I would have to do is throw the "Threads.@threads for i = _____" into every for loop that I could find. To do so, I followed main.jl to see where most of the computations were happening. Main.jl called upon two primary files that contained most of the calculations. The first was disk.jl. It had a few basic serial functions. Cut_cell_solver_explicit.jl on the other hand held dozens of for loops with lots of math. A diagram is shown below.



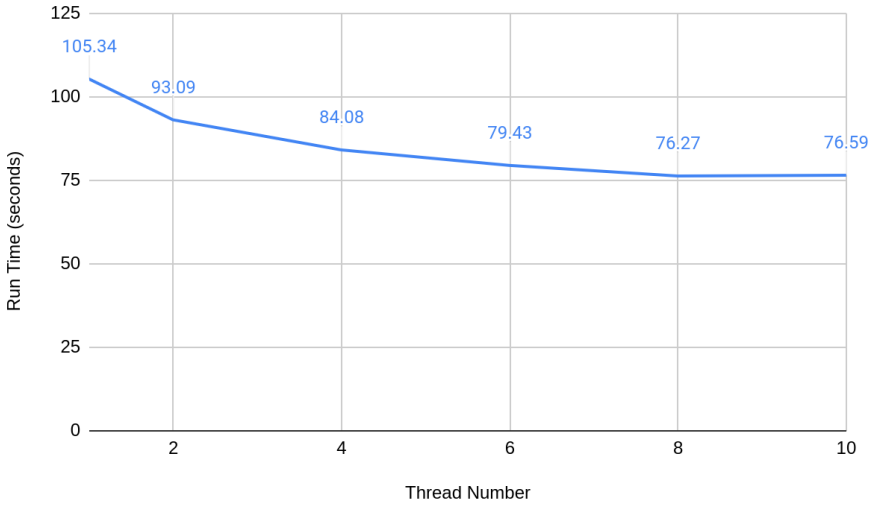
Since nothing inside of disk.jl could be parallelized, all of my time would be spent in cut_cell_solver_explicit.jl. I then did my best to apply a parallelized for loop to each serial loop I could find within the file, completely ignoring any overhead that might be generated. There were a couple that I was not able to parallelize, but the vast majority were. It was time to take a look at the results!

The Results:

The first thing that I did after running was determine that our results looked the same! The original plot was compared with the parallelized plot, and no differences were noted. This was good.



I ran the code on multiple different threads to take a look at how effective my parallelization was. The code was run 2 times on each thread to get an average. The plot below shows the timing results.



The results appear to indicate that there is a significant drop in run time as the number of threads increases. A single thread yielded a run time of 105.34 seconds without any parallelization. Doubling the threads shaved over 10 seconds the first time, and then steadily less as run time approached roughly 76 seconds. This 76 second mark seemed to be a limit that our threads couldn't get past. Amdahl's law was applied to the situation to determine if this really was a limit.

```
Speedup = 1 / [(1 - p) + (p / N)]
```

where:

- `p` is the proportion of the computation that must be executed serially
- `N` is the number of processors used for parallelization”

If we rearrange we can solve for p.

`31.5% of the code is parallelized`

Now with that information we can solve for max possible speed-up (infinite number of threads)

`1.381x faster at ~76 seconds.`

The calculations performed above indicate that our limit that we observed qualitatively is backed up by a quantitative calculation. The question then becomes, is it possible to speed up even more? To determine this I spent some time figuring out where most of the time was being spent.

Can I get Better?

The first potential culprit for me was the code creating the plots in pdf form. I figured this would take a lot of time. However, after the plotting of the graphs took significantly less time than the cut_cell_solver function that was utilized once before each plot was created. This would be where most of the time was spent.

The second spot that I looked at was a large function within the cut_cell_solver. This was an enormous function that to me looked like it would take a lot of time to run. However, as with above, I timed it and it became clear that this was not the culprit. It finished almost instantaneously. Most of the time was being taken up by the third and final culprit.

There is one type of for loop that you just really can’t parallelize, and those are for loops that utilize the previous calculation to calculate the new one. It operates as a sort of “timestep”, in that it is only possible to go forward, not backwards. If you were to try and parallelize the loop, you would be attempting to call upon values that haven’t been calculated yet, thus giving you an error. This was the final spot I looked to try and find areas to speed up. To determine if this was the spot that took the most time, I timed that particular section of code and then modified the number of timesteps. After running the code, it turned out that roughly 70 seconds of the 90 second run time was taken up by this time stepping! This is nearly 80% of the code! While it was great I had figured out where my time was going, it wasn’t great that this area of the code wasn't able to be parallelized. Truly unfortunate.

Another approach?:

Within main.jl there is a single loop that contains the cut_cell_solver function. As a final attempt to speed up the code, I parallelized just this loop and left all other loops serial (as they were called upon within the main.jl loop). The resulting time hovered around 200 seconds. Significantly worse than both serial and parallelized iterations I ran above. This is because the

single loop in main.jl actually contributes almost no time to the overall run time. Thus, adding a parallel loop to it simply adds overhead time without decreasing the time it takes to run the loop.

Conclusion:

Despite my lack of knowledge regarding the math behind the code, I was successfully able to parallelize the code a significant amount. This was possible thanks to Julia's incredibly straightforward multithreading syntax. There were two main takeaways from the process.

1. Throwing threads at the problem doesn't fix it.

This was taught to us in class but I didn't really understand the weight this carries until the project. We can clearly see in the time v threads plot that increasing the number of threads past a certain point does next to nothing. If only 30 percent of your code is parallelized, there is only so much you can reduce the run time by. If you really want to speed your code up, you need to figure out ways to increase the percent of your code that is parallel. This brings us to the second point.

2. If you don't fully understand the code, you can only get so far.

I was pretty easily able to throw parallel loops around almost all of the code. However, there were a few loops that I was unable to fully parallelize. This meant that there were a few seconds that were probably spent performing calculations that could have been parallelized if I had learned how the code worked.

3. You just can't get perfectly parallel sometimes.

Probably the hardest lesson from this, but sometimes the code simply cannot be parallelized as much as you may want it to be. You may be able to determine a faster algorithm, or a different way of doing things, but at the end of the day, there are limits to how fast you can get your code. As we saw in this project, the timestepping loop wasn't able to be parallelized due to its forward moving nature. Unfortunately, this also took up most of the time of the code, and so pretty significantly limited how much I could parallelize from the get go.

Thoughts:

Overall I actually had a lot of fun with this project. Open ended questions give me the ability to play around with the code, and learn for myself with a real world problem. I was a particular fan of the fact that I worked on complex code. This allowed me to truly get a grasp on some of the limitations that parallel programming encounters in the real world. Class discussions can only get so far. Applying to actual, complex code makes a world of difference in how much I learn. Really enjoyed class this semester. Thanks for all the help and I look forward to applying what I've learned in my internship this summer!