Final Project Report

CS 5030 – High Performance Computing

Streamlines

Shared Memory

For this implementation, I used C++11 threads. My reason for this choice is to have the algorithm look as close as possible to the GPU implementation. The algorithm takes the number of threads and divides the number of streamlines to calculate (600) evenly between each thread (note that the number of threads should divide 600).

For storing the results, an array global to the threads is allocated with the expected total number of floats to be calculated across all of the threads. This array is initialized with values of -1. This choice is made as we don't expect any negative values in this array. Then with each calculation, a thread populates the correct spot of this array as long as the points calculated do not leave the bounds of the vector field.

Once the threads are finished, the main process loops through this array, writing 3 floats at a time to file: line_id, coordinate_x, coordinate_y. If a spot where line_id is expected is equal to -1, that line is skipped as it indicates that a streamline left the bound of the vector field.

For scaling, I timed my implementation with 1,2,3,4 and 6 threads. I only tracked the time that the threads spent working; I excluded the time spent reading in from and writing to file. These are the results (average of 10 runs):

Number of Threads	1	2	3	4	6
Time (ms)	39.4	21.4	17.7	11.1	8.8

Outliers ignored: on 1 thread, there was a run that took 91 ms; on 3 threads, a run took 97 ms;

We see consistent improvement in the timing as the number of threads increases. It is notable, however, that across the various runs, there was a lot of variability. The longest run on 6 threads was longer than the average of 4 threads, for example.

Distributed Memory

For the distributed memory implementation, I used MPI. However, I misunderstood this concept in my implementation as I had all of the processes have access to the entire vector field. Beyond that, the algorithm here was identical to the one used in shared memory, replacing threads with processes. There is however, one important; instead of having one large array that all of the processes write to, each process had its own local array where it stored its results. This local array was then sent to the main process where the results were written to file, in process order. In this regard, my implementation has a small measure of distributed memory.

I ran the program on 2,4,8, and 15 processes. These are the results:

Processes	2	4	8	15
Time (ms)	20.8	14.3	12.6	13.6

GPU

For the GPU implementation, I used CUDA. Here again, the algorithm was identical to the shared memory implementation save that instead of a thread working on several lines, each thread does only one line. For the scaling study, I experimented with different block sizes. Given the design of my program, I only used one. I then experimented with 3 differently sized blocks: 100, 200, 300, 400, 500, and 600. The given values are the averages of 10 runs.

Tile size	100	200	300	400	500	600
Time (ms)	8	5	4	2	3	2

Interestingly, the time across all 10 runs for block size was the same, no variability. Probably related to that is in the fact that the output for the time had only 1 significant digit. I am unsure why this happened. The most interesting result here is that the tile size of 400 turned out to be as fast as 600. This is surprising since at that size, each thread will do more than line whereas in the case of 600, each thread would do only one line. I attribute this to the natural variability in time that comes with comuters.

Comparisons

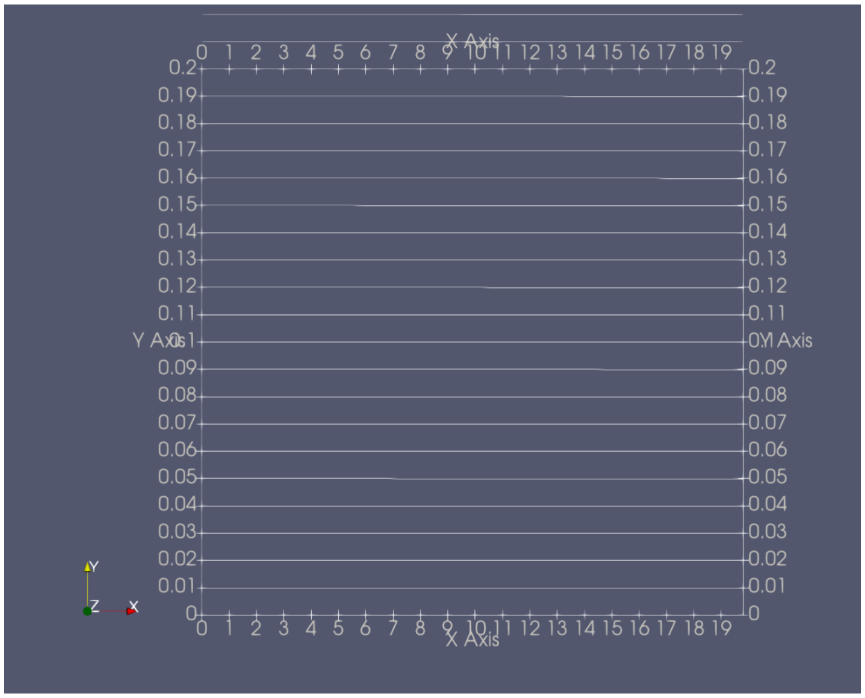
Not surprisingly, the GPU was by far the fastest in making these calculations. The reason is the number of threads – the GPU could calculate hundreds of streamlines at the same time while the others had to do over 100 for each thread / process.

I found it interesting that shared memory experienced the largest speedup. Comparing it to MPI, we see that shared memory was slower on 1 thread but faster on 6 than MPI with 8 processes. This is what I expect as in shared memory, the threads / processes do not need to communicate with each other.

It is important to note that MPI would likely perform much better on a much larger dataset, where it wouldn't necessarily fit on one process. This would require communication between processes that isn't entirely necessary with this data (as seen in my short-sighted implementation).

Visualizations

There must have been an error in my Runge-Kutta algorithm as I only got horizontal lines. The following is the best image I could come up with. In Paraview, to view all of lines was just a long vertical bar that was unintelligible. What is shown here is zoomed version of the data. The vertical white lines are the streamlines.



Problems

The most notable problems with my implementations lie within my algorithm. Somehow, I only got straight, horizontal lines. I combed through my code dozens of times and was unable to figure out why. It doesn't help that I don't fully understand how this algorithm works to begin with; I just copied the logic from resources I was given.

I also noticed that in my cuda implementation, the line IDs were not always printed as integers. Rather, they were printed as ~0.0001 less than the intended integer. I am again at a complete as to why this happened. My best guess is that it has something to do with my memory allocation being slightly off.

My last issue came in the form of the visualization. I could not figure out how to configure the Paraview program to display the data is a visually appealing way.