CS 5030 HPC
Final Project Report

Manuel Santana: A02226822

December 9, 2021

This project was the streamlines project. Using the given vector field it uses a fourth order rugge-kutta method to compute streamlines, with one streamline seeded at every row, making 600 streamlines in total. The streamlines were computed with time steps of 0.1 and ran for 12,000 iterations each. If a streamline left the vector field in it was assigned the last in bound value for the rest of its iterations. Three implementations were written:

1. Shared Memory: The approach to the shared memory implementation was to parallelize over the rows. OpenMP was used to parallelize the loop over which the each seed's streamline was computed.
   To compile (use g++ 6 or greater):

   ```
   g++ -o [myExec] -fopenmp StreamLinesOMP.cpp
   ```

   To run:

   ```
   ./myExec [numThreads]
   ```

2. GPU: The GPU approach was similar to the shared memory approach. One thread was assigned to each seed and computed the entire streamline.

   To compile:

   ```
   nvcc -o [myExec] -fopenmp StreamLinesGPU.cpp
   ```

   To run:

   ```
   ./myExec
   ```

3. Distributed Memory: For the distributed memory implementation each process was send a block of the vector field with an equal number of rows. Then each process was responsible for computing the streamline where the seeds started at that part of the vector field. If one streamline went off the local vector field values, the process messaged the process who had the needed data and received that data. Each process was only allowed to have the vector field data from one other process at a time. All processes were forced to check for messages after they finished their computation until all processes were finished computing.

   To compile (please use MVAPICH2 for node counts higher than 15):

   ```
   mpic++ -o [myExec] StreamLinesMPI.cpp
   ```

   To run:

   ```
   mpiexec -np [numProcesses] ./myExec
   ```

   Note that in MPI_Scatter was used to distribute the vector field, and so the number of processes should divide the number of rows, which is 600.

Figure 1 is the visualization generated in Paraview of the streamlines, with red as the x direction and yellow as the y direction.

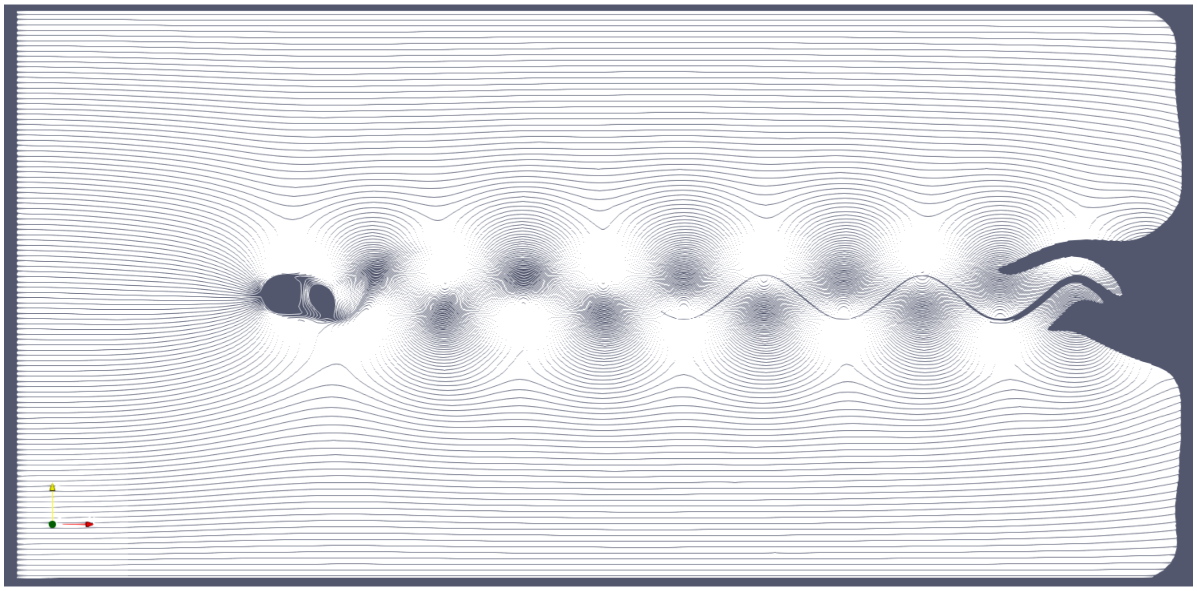

Figure 1: Stream Line Visualization

Scaling Studies

Here we discuss the scaling results starting with OpenMP. We note that we don't exactly double the number of threads starting at 15 so that the number of threads evenly divides 600, the number of rows. This made implementation easier for MPI. Table 1 and figure 2 give the results for OpenMP. These runs were made on a kingspeak compute node.

Number of Threads	Run Time	Speedup	Efficiency
1	4.7086	1.0000	1.0000
2	2.3590	1.9960	0.9980
4	1.5764	2.9870	0.7467
8	0.5909	7.9690	0.9961
15	0.3146	14.9652	0.9977
30	0.2836	16.6007	0.5534
60	0.2953	15.9435	0.2657

Table 1: OpenMP Run Details

The first thing to notice is that as expected the run time went down and the speed up went up as threads were increased, except in the case of 60 threads. This could be because the timing were taken on one node which might not support 60 threads, or because the overhead of creating and joining threads outweighed the benefit. The latter reason is most likely why the efficiency wend down for the 60 threads. It is interesting to see how the efficiency went down for 4 threads, and that could be because it was a slow run, and more tests could show that on average the efficiency is higher for 4 threads.

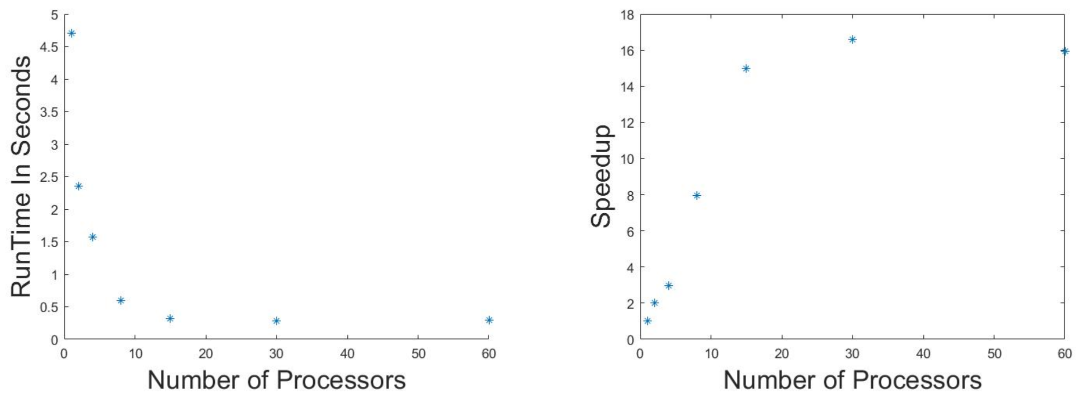

Figure 2: OpenMP RunTimes and Speedup

Next the GPU program was run ten times with 50 threads per block on a Tesla k80 GPU on the notchpeak cluster. The average time was 0.22467 seconds, with a fastest time of 0.18921 seconds, and a slowest time of 0.28689 seconds. This gave an average speed up of 20.9578 over the OpenMP single thread implementation. Changing threads per block did not cause a significant change in the run time. Tiling was not possible in an effective manner with out assuming some a priori knowledge of the structure of the streamlines, and so tiling was not done.

Finally we have the MPI run times and plots. The runs were made on a kings peak compute node.

Number of Processes	Run Time	Speedup	Efficiency
1	3.3242	1.0000	1.0000
2	1.7362	1.9147	0.9573
4	0.9227	3.60267	0.9006
8	0.5127	6.48327	0.8104
16	0.3880	8.5686	0.5712
32	1.2197	2.7254	0.0908
64	58.2405	0.0571	0.0010

Table 2: Timings for MPI

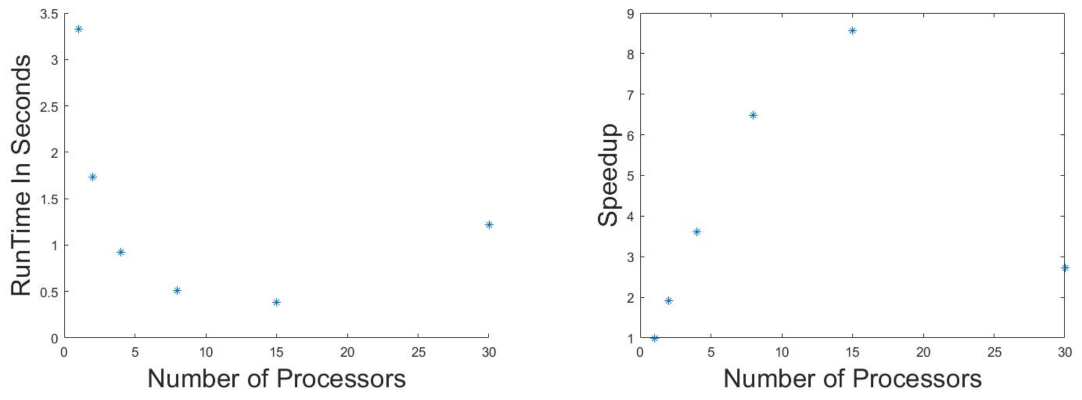

Figure 3: MPI RunTimes and Speedup (not including 60 processes)

The first thing to notice is that as expected up to fifteen processes the run time goes down and the speedup goes up, and the efficiency goes down as you have more processes due to more communication. But then for thirty processes the run time goes up. One might ask why this happens. Having the program print out when communications were happening I found that the processes computing the middle streamlines did a lot of communication. If you look at the visualization in the middle there is a lot of movement in y direction in the streamlines. With thirty processes each process has only a very small amount of vector field data, and so it has to change the cache many times as the streamlines move up and down. Therefore the amount of communication goes up drastically with those processes. Additionally those processes computing the streamlines in the middle will have to communicate with each other a lot making it worse. For example if 14 is waiting on 15, and 13 is waiting on 14 then there are two processes waiting on 15. This problem is exacerbated with 60 processes, so much so that 60 processes is not included on the graphs because it makes the graphs unreadable.