# Parallel Programming HW1 Writeup

Andrew McCarthy

- Overview:

This program is designed to print out the shortest path matrix given an input file with an N x N grid. The first line of the input file should be a number representing the size of the matrix. Then, there should be N lines each with N numbers (separated by a tab) each representing the edge weight to go from node i to node j. The program can run in serial or parallel mode. To enable parallel mode, provide a number representing the maximum number of threads as the first argument and a string representing the file with the input data as the second argument. To enable serial mode, just provide the file with the input data.

- Details:

The project code is in the fw.c file. The testing.c file is performance testing code. It compares times between the serial and parallel implementations using set max threads and vertices values. It prints the results of these tests to stdout. Running "make" will compile the code and create the executables. The executables are named 'fw' and 'testing'. Running "make clean" will delete all the executables and *.o files. For the serial implementation, make the first and only argument a string representing the file with the input data. For the parallel implementation, make the first argument a number representing the number of strings, and the second argument a string representing the file with the input data.

- Design:

The program uses barriers and pthreads to implement the parallel version. First, the data from the input file is loaded into a one dimensional array that is used to represent the N x N matrix. Each N numbers in the one dimensional array represents one line in the two dimensional input array. Next, all the threads and the barrier are created. Next, each thread starts work on the thrice deep loop with the i loop iterations being divided evenly between the threads. The barrier is used to sync each thread between k loop iterations to ensure all threads have the same value for k. Once the threads are done, they are all joined and the resulting distances matrix is printed to the output file. The serial implementation of this code works the same as the parallel implementation but without using any threads or barriers. In my design document, I planned to just divide the final j loop among the threads. However, after the class discussion, I decided to divide the i and j loops among the threads as it was simpler to implement. Also, I originally planned to assign portions of the loop to threads dynamically while the algorithm was running. While coding however, I decided statically assigning portions of the loop before running the algorithm was easier. The algorithm used is the one outlined in the wikipedia article provided to us. In addition, I test for invalid matrices of size 0.

- Performance Testing:

For performance testing, I created randomly generated input data using every value for vertices count provided to us in the project write up {16, 32, 64, 128, 256, 512, 1024}. Then, I ran the parallel and serial versions of the algorithm using every value of max thread count also provided to us in the project description {2, 4, 8, 16, 32, 64}. I used the serial output to ensure the parallel version was generating correct input data. I used the stopwatch data structure provided to us to calculate the times in the serial and parallel versions. Finally, I printed the results of these tests to stdout. Also, in the beginning I ran the parallel implementation with 1 thread and the serial implementation using every value for vertices count to calculate the overhead incurred by the parallel version.

- Correctness Testing:

To ensure the serial implementation was correct, I ran a few tests with with input data that had verified output data. Once I was sure the serial implementation was creating correct output data, I ran the test program using the serial version as a correctness checker. I ran the test program on the SLURM cluster and on my personal machine (i9-9820X 10 core 20 threads). I found that the results from SLURM somewhat lined up with my performance hypothesis with quite a few notable exceptions.

Test matrix 1:

```
7
0       3       6       1000000 1000000 1000000 1000000
3       0       2       1       1000000 1000000 1000000
6       2       0       1       4       2       1000000
1000000 1       1       0       2       1000000 4
1000000 1000000 4       2       0       2       1
1000000 1000000 2       1000000 2       0       1
1000000 1000000 1000000 4       1       1       0
```

Verified output:

```
7
0       3       5       4       6       7       7
3       0       2       1       3       4       4
5       2       0       1       3       2       3
4       1       1       0       2       3       3
6       3       3       2       0       2       1
7       4       2       3       2       0       1
7       4       3       3       1       1       0
```

Test matrix 2:

```
8
0      576    409    409    839    425    481    505
850    0      222    570    182    519    524    165
295    106    0      881    546    8      206    389
462    507    354    0      156    494    959    497
784    298    660    359    0      58     69     550
834    549    54     683    122    0      623    216
993    146    380    287    252    612    0      185
611    817    573    72     675    278    579    0
```

Verified output:

```
8
0      515    409    409    539    417    481    505
517    0      222    237    182    230    251    165
295    106    0      296    130    8      199    224
462    371    268    0      156    214    225    410
407    215    112    326    0      58     69     254
349    160    54     288    122    0      191    216
659    146    364    257    252    310    0      185
534    438    332    72     228    278    297    0
```

Test matrix 3:

```
4
0       5       1000000 10
1000000 0       3       12
2       17      0       1
1000000 1       1000000 0
```

Verified output:

```
4
0      5      8      9
5      0      3      4
2      2      0      1
6      1      4      0
```

- Performance Testing Results:

I ran performance and parallel overhead testing on the SLURM cluster and my personal machine. The results, posted to stdout, are listed below.

--SLURM results

overhead with 16 vertices: serial 0, parallel 0
overhead with 32 vertices: serial 0, parallel 0
overhead with 64 vertices: serial 0, parallel 0
overhead with 128 vertices: serial 4, parallel 4
overhead with 512 vertices: serial 244, parallel 252
overhead with 1024 vertices: serial 1863, parallel 1883
For 16 vertices and 2 threads, serial time was 0 and parallel time was 0
For 32 vertices and 2 threads, serial time was 0 and parallel time was 0
For 64 vertices and 2 threads, serial time was 0 and parallel time was 0
For 128 vertices and 2 threads, serial time was 4 and parallel time was 3
For 512 vertices and 2 threads, serial time was 242 and parallel time was 126
For 1024 vertices and 2 threads, serial time was 1870 and parallel time was 953
For 16 vertices and 4 threads, serial time was 0 and parallel time was 0
For 32 vertices and 4 threads, serial time was 0 and parallel time was 0
For 64 vertices and 4 threads, serial time was 0 and parallel time was 0
For 128 vertices and 4 threads, serial time was 4 and parallel time was 2
For 512 vertices and 4 threads, serial time was 242 and parallel time was 69
For 1024 vertices and 4 threads, serial time was 1867 and parallel time was 493
For 16 vertices and 16 threads, serial time was 0 and parallel time was 1
For 32 vertices and 16 threads, serial time was 0 and parallel time was 1
For 64 vertices and 16 threads, serial time was 0 and parallel time was 3
For 128 vertices and 16 threads, serial time was 4 and parallel time was 6
For 512 vertices and 16 threads, serial time was 243 and parallel time was 117
For 1024 vertices and 16 threads, serial time was 1860 and parallel time was 641
For 16 vertices and 32 threads, serial time was 0 and parallel time was 2
For 32 vertices and 32 threads, serial time was 0 and parallel time was 3
For 64 vertices and 32 threads, serial time was 0 and parallel time was 6
For 128 vertices and 32 threads, serial time was 4 and parallel time was 13
For 512 vertices and 32 threads, serial time was 243 and parallel time was 128
For 1024 vertices and 32 threads, serial time was 1865 and parallel time was 393
For 16 vertices and 64 threads, serial time was 0 and parallel time was 4
For 32 vertices and 64 threads, serial time was 0 and parallel time was 7
For 64 vertices and 64 threads, serial time was 0 and parallel time was 12

For 128 vertices and 64 threads, serial time was 4 and parallel time was 23
For 512 vertices and 64 threads, serial time was 246 and parallel time was 133
For 1024 vertices and 64 threads, serial time was 1865 and parallel time was 458

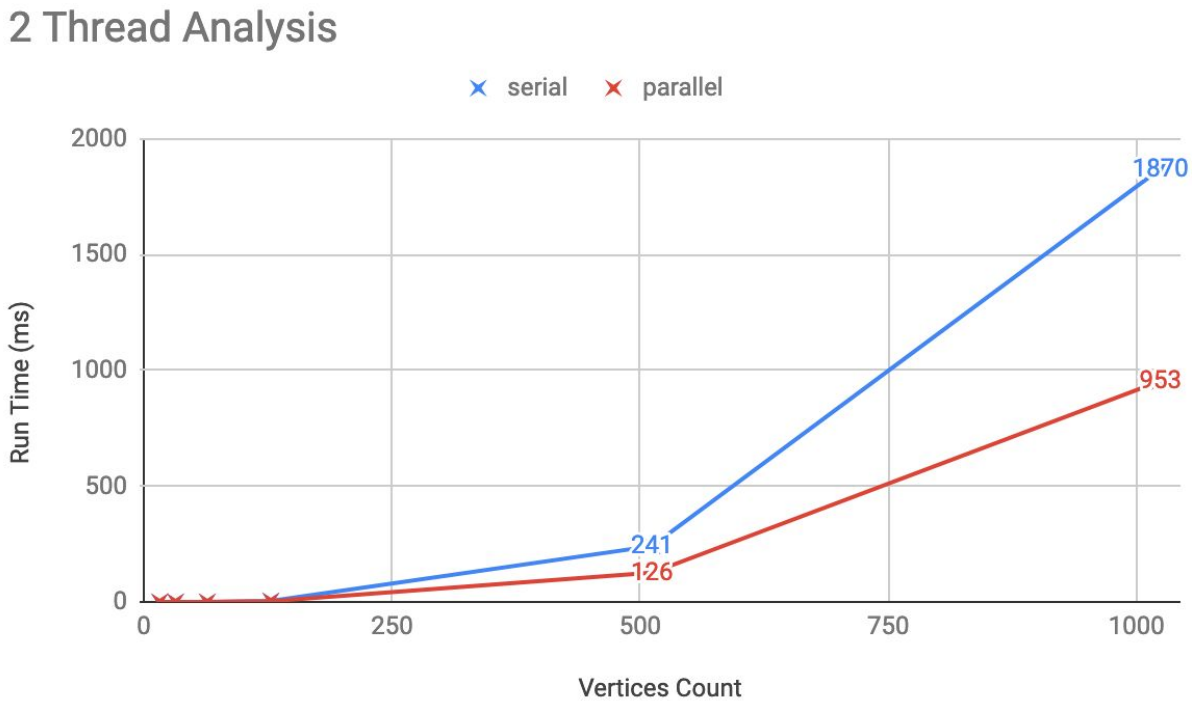
--i9-9820X 10 core 20 threads

overhead with 16 vertices: serial 0, parallel 0
overhead with 32 vertices: serial 0, parallel 0
overhead with 64 vertices: serial 1, parallel 1
overhead with 128 vertices: serial 7, parallel 9
overhead with 512 vertices: serial 140, parallel 170
overhead with 1024 vertices: serial 878, parallel 1066
For 16 vertices and 2 threads, serial time was 0 and parallel time was 0
For 32 vertices and 2 threads, serial time was 0 and parallel time was 0
For 64 vertices and 2 threads, serial time was 0 and parallel time was 1
For 128 vertices and 2 threads, serial time was 2 and parallel time was 5
For 512 vertices and 2 threads, serial time was 153 and parallel time was 88
For 1024 vertices and 2 threads, serial time was 883 and parallel time was 551
For 16 vertices and 4 threads, serial time was 0 and parallel time was 0
For 32 vertices and 4 threads, serial time was 0 and parallel time was 0
For 64 vertices and 4 threads, serial time was 0 and parallel time was 0
For 128 vertices and 4 threads, serial time was 2 and parallel time was 3
For 512 vertices and 4 threads, serial time was 126 and parallel time was 60
For 1024 vertices and 4 threads, serial time was 909 and parallel time was 359
For 16 vertices and 16 threads, serial time was 0 and parallel time was 0
For 32 vertices and 16 threads, serial time was 0 and parallel time was 1
For 64 vertices and 16 threads, serial time was 0 and parallel time was 2
For 128 vertices and 16 threads, serial time was 2 and parallel time was 4
For 512 vertices and 16 threads, serial time was 126 and parallel time was 52
For 1024 vertices and 16 threads, serial time was 885 and parallel time was 131
For 16 vertices and 32 threads, serial time was 0 and parallel time was 1
For 32 vertices and 32 threads, serial time was 0 and parallel time was 2
For 64 vertices and 32 threads, serial time was 1 and parallel time was 4
For 128 vertices and 32 threads, serial time was 7 and parallel time was 11
For 512 vertices and 32 threads, serial time was 153 and parallel time was 74
For 1024 vertices and 32 threads, serial time was 875 and parallel time was 391
For 16 vertices and 64 threads, serial time was 0 and parallel time was 2
For 32 vertices and 64 threads, serial time was 0 and parallel time was 4
For 64 vertices and 64 threads, serial time was 0 and parallel time was 8

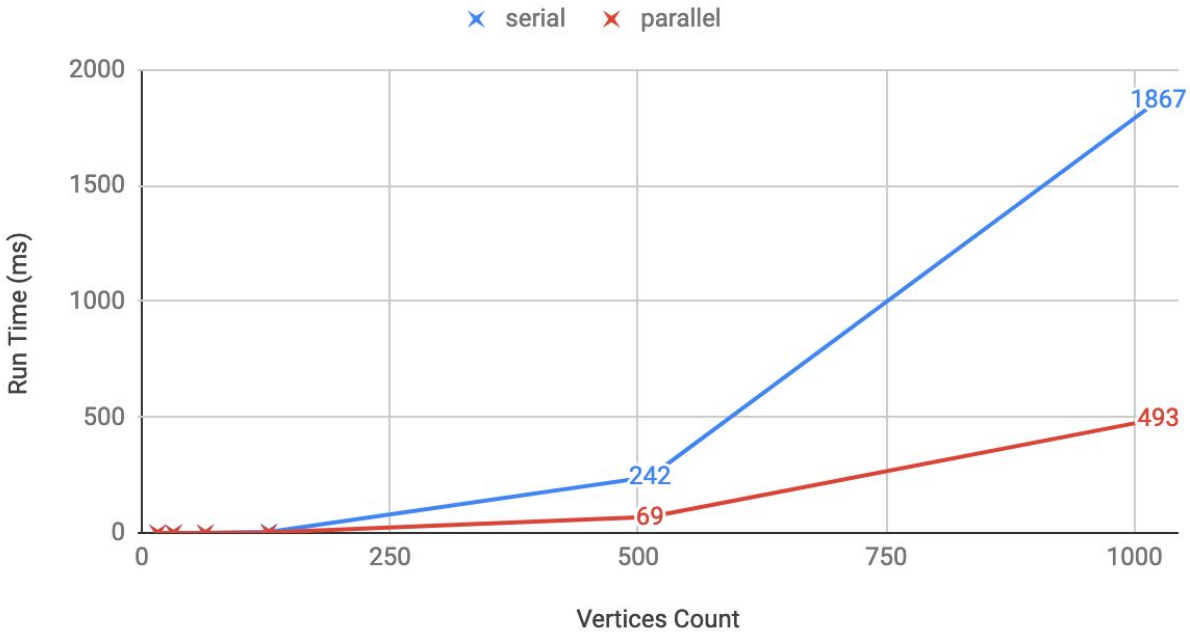
For 128 vertices and 64 threads, serial time was 5 and parallel time was 20
For 512 vertices and 64 threads, serial time was 131 and parallel time was 85
For 1024 vertices and 64 threads, serial time was 879 and parallel time was 432

- Testing Analysis:

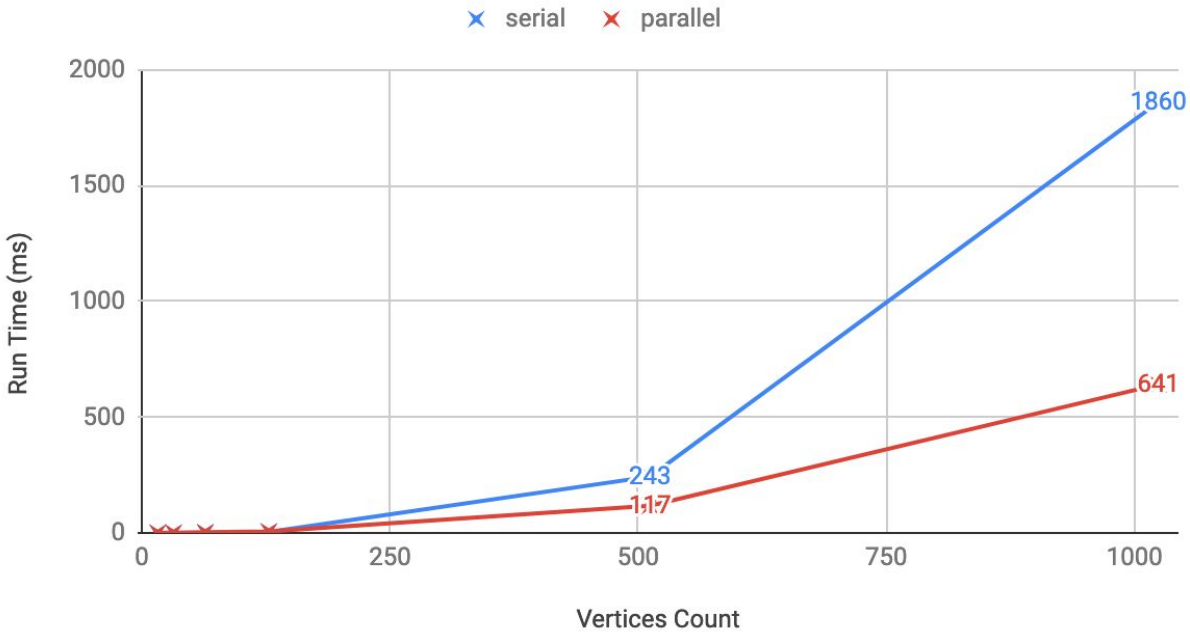
Posted below are graphs of the performance testing results.



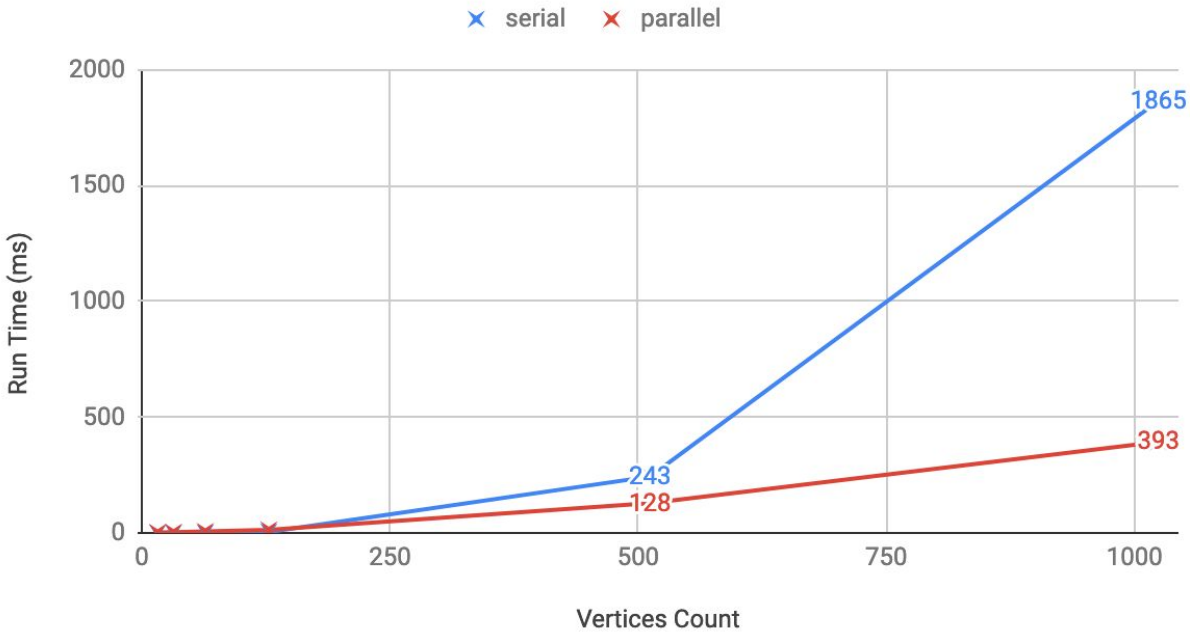
## 4 Thread Analysis



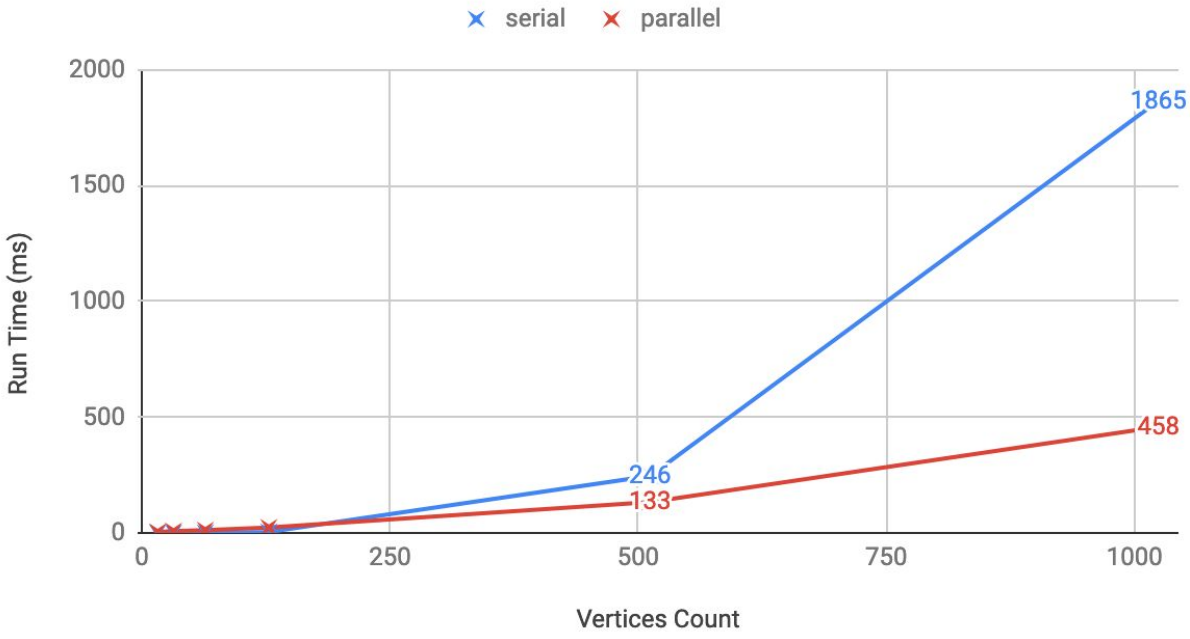

## 16 Thread Analysis



## 32 Thread Analysis




## 64 Thread Analysis



In my overhead testing, I found that the parallel version was definitely slower than the serial version, but not in any way that would affect my analysis. Generally, the parallel version added an extra 10-20 ms of run time.

In my design document, I predicted that performance would scale by a factor of "max threads" and that performance increases would only be noticeable at high vertices counts. This is true for two and four threads at a vertices count of 1024 (and somewhat at a vertices count of 512). The parallel version is about two times and four times faster respectively. For 16, 32, and 64 threads however, the performance increase was more unusual. They didn't follow any noticeable pattern and 16 threads was actually slower than 4. I suspect this is because at higher thread counts with a small number of vertices, the performance that is gained is negligible compared to the same execution with a smaller number of threads. In addition, the overhead of initializing all the threads and using the barrier outweighs any performance benefit they incur. Thus, it seems that using any more than 4 threads at this low of a vertices count is not worth it.