Project Assignment 04 Report

CS415

5-3-2017

By: Jervyn Suguitan

Table of Contents:

Description of Assignment

Project 04's main purpose was to teach students about matrix multiplication and how to parallelize the code. This report will talk about running matrix multiplication sequentially and parallelized, how to code each version, and discussion based on the timings from the code. The timing function that was used to time calculation for this project was MPI_Wtime.

Part 1: Sequential

The first part of the project tasks students in figuring out how to code matrix multiplication sequentially. For the sequential code of the matrix multiplication, the first thing to do is to create three 2-D arrays which act as the 3 matrices needed for the multiplication. In the getData function two of the matrices will be filled with numbers and act as the matrixA and matrixB which will be multiplied. These two matrices get their numbers from a file specified in the command line. The first matrix will get its data from matrixA file and the second will get its data from matrixB file. The third matrix is used to hold the result of multiplying matrix A and matrix B. The three matrices are put into a matrix multiplication function in which the actual matrix multiplication occurs. All matrices are of the same size, as in the width and length of the matrices are equal. The sizes of the matrices are all multiples of 120 due to the fact that for the parallel code, also uses a perfect square amount of processors (4, 9, 16, 25). The matrices are multiples of 120 as 120 can be divided by those perfect square amounts. The runtime of matrix multiplication is O(n^3).

Figure 1, Fig. 2 and Fig. 3 showcase the runtime execution of the sequential matrix multiplication code with multiple different matrix sizes. The only difference between Fig. 1, Fig. 2 and Fig. 3 is that Fig. 1 includes the trials with all matrix sizes while Fig. 2 and Fig. 3 showcase a close look to each set of tests. These matrix sizes were chosen to show times over a large variety of sizes. These graphs show that as more numbers that are needed to be sorted, the longer it takes to sort.

From Fig. 1, Fig.2, and Fig. 3 we can see that the runtimes for the same size matrices are usually very similar. Generally, these times do not vary too much is because there is no message passing between master and slave which takes up time. There is no message passing since this code is sequential, thus there is no need to use multiple processors. To say that this code uses only one core when running the program we use a "-n1".

Another attribute of these figures is that the run times increase in an exponential rate as the sizes of the matrices increase. This occurs since there are more calculations for larger matrices. For specifically Fig. 1 and Fig. 3, near the larger matrices around 2640, there is a large difference between test runtimes. These differing runtimes could have occurred due to network congestion. For example during times when the head node is not busy, running the sequential matrix multiplication code with a 2640 matrix is around 3 minutes and 10 seconds, while the average time for this matrix is 3 minutes and 30 seconds which can be found in Table 1 below.

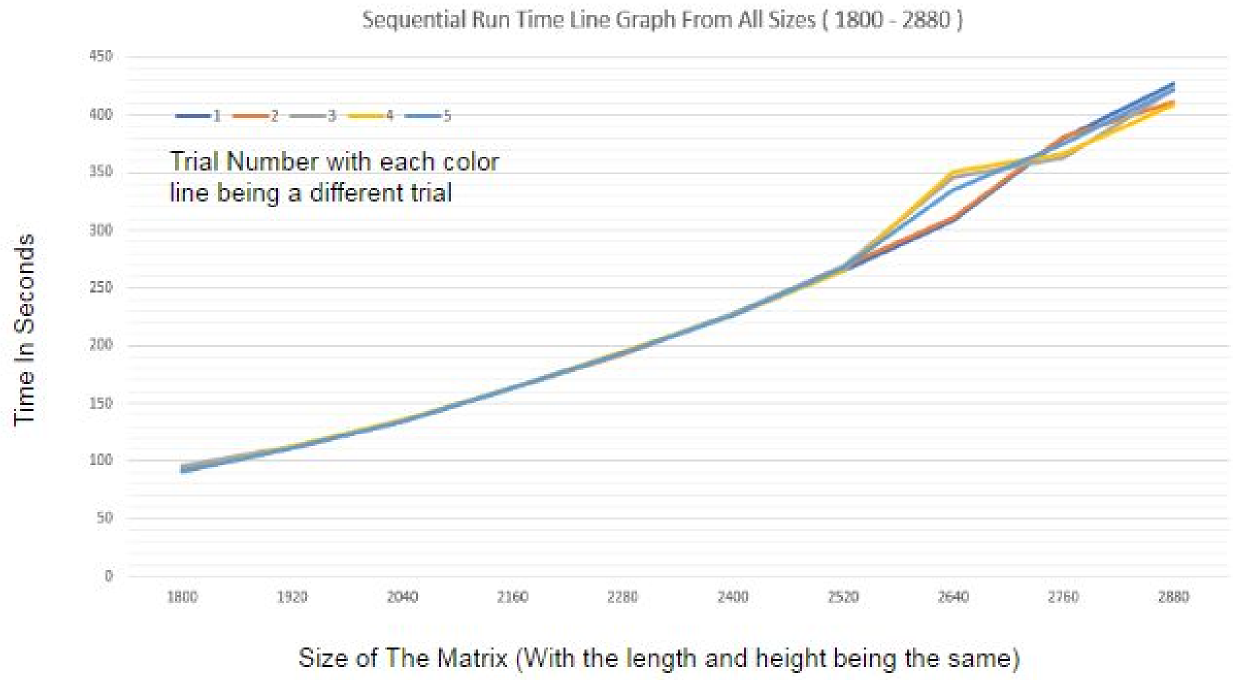
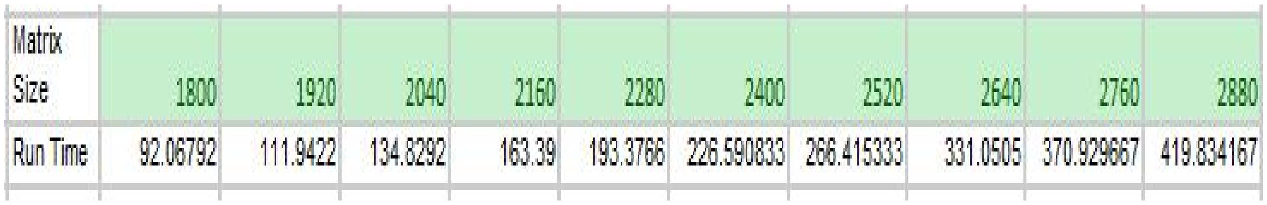


Fig. 1: Graphs that showcase the sequential times for different matrix sizes when doing matrix multiplication. The Y axis shows the time in seconds and the x-axis shows the matrix size. This graph includes times for 2880 by 2880 sized matrix. The times do not vary large amounts since everything is sequential thus there can not be a large speed up. Speed up or slow down may occur due to the processor speed which could be different for every trial. Some of the trials are hard to see as they are behind other lines.

Table 1 shows the average sequential time for matrix multiplication. The ranges of the numbers inside matrixA and matrixB are from 1 to 50. The runtimes of the of these matrix sizes are from one minute to seven minutes. I did this so I could see the efficiency and speed up for very large matrices later on the project.

Table 1: Chart that showcases the average times of different sizes of matrices. These run times are sequential thus it uses only one core.

Table that Shows the Average Runtimes of Different Matrix Sizes: Sequential



Matrix Size	1800	1920	2040	2160	2280	2400	2520	2640	2760	2880
Run Time	92.06792	111.9422	134.8292	163.39	193.3766	226.590833	266.415333	331.0505	370.929667	419.834167

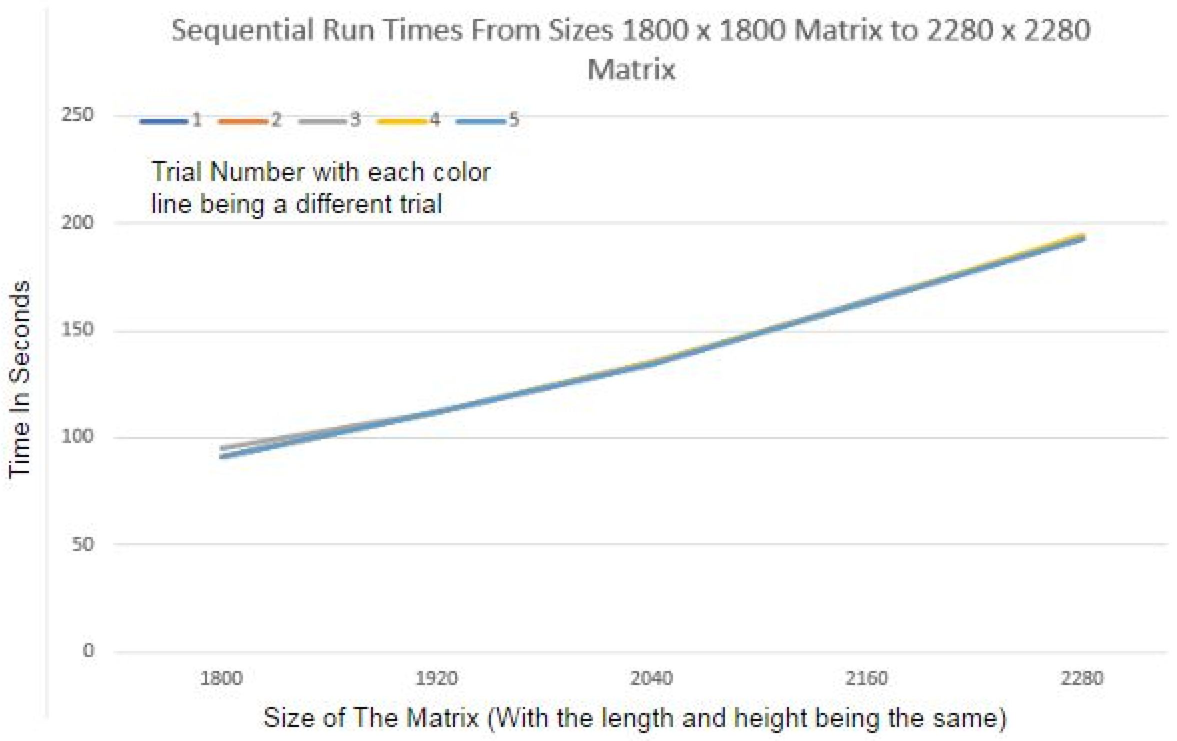


Fig. 2: Graphs that showcase the sequential times for different matrix sizes when doing matrix multiplication. The Y axis shows the time in seconds and the x-axis shows the matrix size. This graph includes times for 1800 to 2280 sized matrix. The times do not vary large amounts since everything is sequential thus there can not be a large speed up. Speed up or slow down may occur due to the processor speed which could be different for every trial. Some of the trials are hard to see as they are behind other lines.

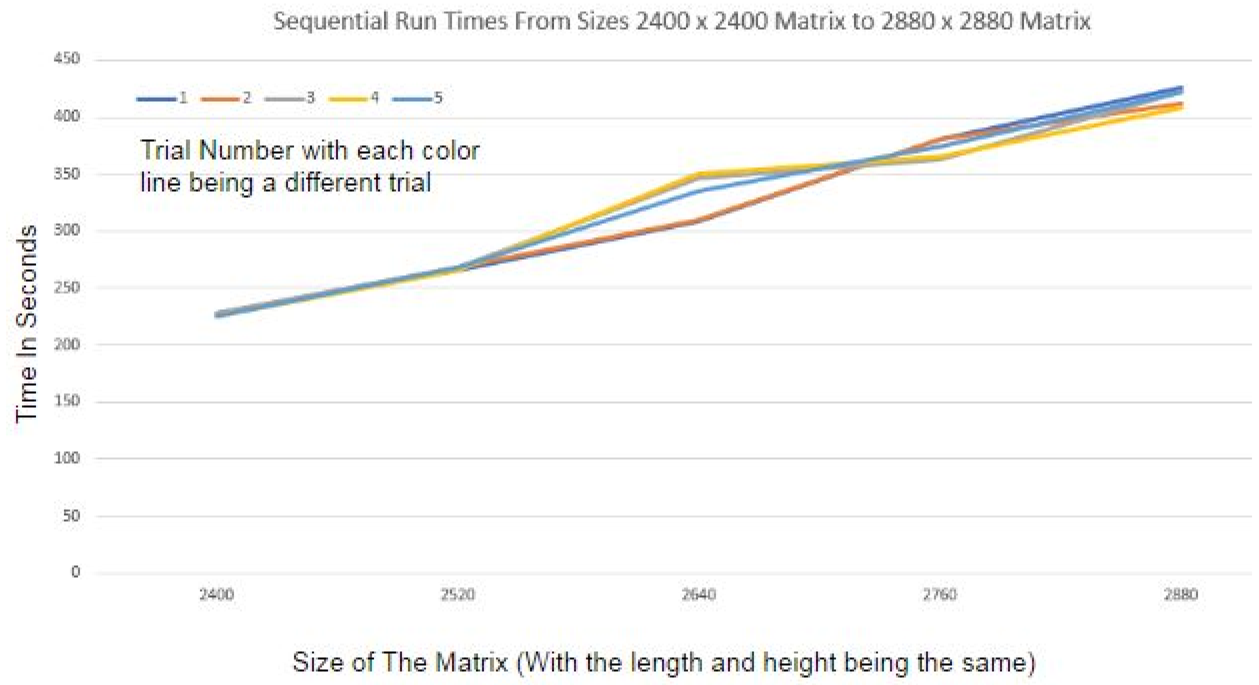


Fig. 3: Graphs that showcase the sequential times for different matrix sizes when doing matrix multiplication. The Y axis shows the time in seconds and the x-axis shows the matrix size. This graph includes times for 2400 to 2880 sized matrix. The times do not vary large amounts since everything is sequential thus there can not be a large speed up. Speed up or slow down may occur due to the processor speed which could be different for every trial. Some of the trials are hard to see as they are behind other lines.

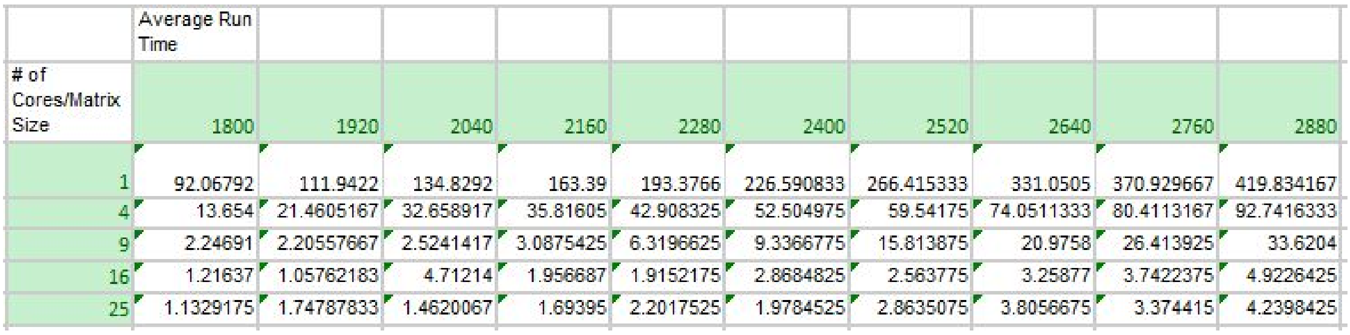
Problems Encountered in Sequential

I had minor problems modularizing everything. I wanted to create functions to read in the files into the matrices. I also printed things improperly which reversed the final matrix order.

Part 2: Parallel Matrix Multiplication

The next part of the projects tasks students to parallelize their sequential code. The idea for the parallel code is somewhat similar to the sequential code with one major difference which is Cannon's algorithm. Cannon's algorithm is just a way to do matrix multiplication on multiple processors. The parallel code starts out similarly to the sequential code in which you put three 2-D arrays into the getData function. In the getData function you fill matrix A and matrix B with the two files which are from the command line. The first matrix will get its data from matrixA file and the second will get its data from matrixB line. After the matrices are filled the master sends these two matrices (matrixA and matrixB) to the slaves through MPI_Broadcast. After that, all processors create 3 2-D matrices. These new matrices act as sub matrices that will swap around to do Cannon's algorithm correctly. To figure out which processor is working on what area of the main matrices, I have a findWorkingRows function that finds out that information. Next, all processors figure out which area of matrixA and matrixB they will work from, they will copy those matrices onto their own sub matrices. Once all processors have their correct sub matrix they all reach a MPI_Barrier call and then all the matrices are moved depending on their processor number (myRank). This initial movement is important to have Cannon's algorithm to work. After this initial matrix movement, the matrices are multiplied and the matrices are moved again. After moving the matrix sqrt(number of Processors) amount of times, the matrix multiplication is completed. And the run time is taken after all the multiplication occurs.

Table 2: Table that shows the average run times of different matrices on different number of processors in **Seconds.**



	Average Run Time									
# of Cores/Matrix Size	1800	1920	2040	2160	2280	2400	2520	2640	2760	2880
1	92.06792	111.9422	134.8292	163.39	193.3766	226.590833	266.415333	331.0505	370.929667	419.834167
4	13.654	21.4605167	32.658917	35.81605	42.908325	52.504975	59.54175	74.0511333	80.4113167	92.7416333
9	2.24691	2.20557667	2.5241417	3.0875425	6.3196625	9.3366775	15.813875	20.9758	26.413925	33.6204
16	1.21637	1.05762183	4.71214	1.956687	1.9152175	2.8684825	2.563775	3.25877	3.7422375	4.9226425
25	1.1329175	1.74787833	1.4620067	1.69395	2.2017525	1.9784525	2.8635075	3.8056675	3.374415	4.2398425

Table 2 above showcases the runtimes of different matrices on different number of processors in a table form. Based on the table you can see the general trend that as the size of the matrix increases so does the runtime. The sequential run times are also much larger than any of the parallel run times. At larger end of the matrix sizes, we can see that as more cores are used the smaller the run time. However at the 16 and 25 cores there times are very similar even with the 9 core difference, this occurs mainly due to the message passing that occurs. My parallel code breaks matrixA and matrixB into a sqrt(# of processors) number of sub matrices. When passing these sub matrices around for Cannon's algorithm I send a row at a time which is makes message passing occur many times throughout execution. Below Fig. 3 shows a visual representation of Table 2, average run time of matrix multiplication over different matrix sizes and cores.

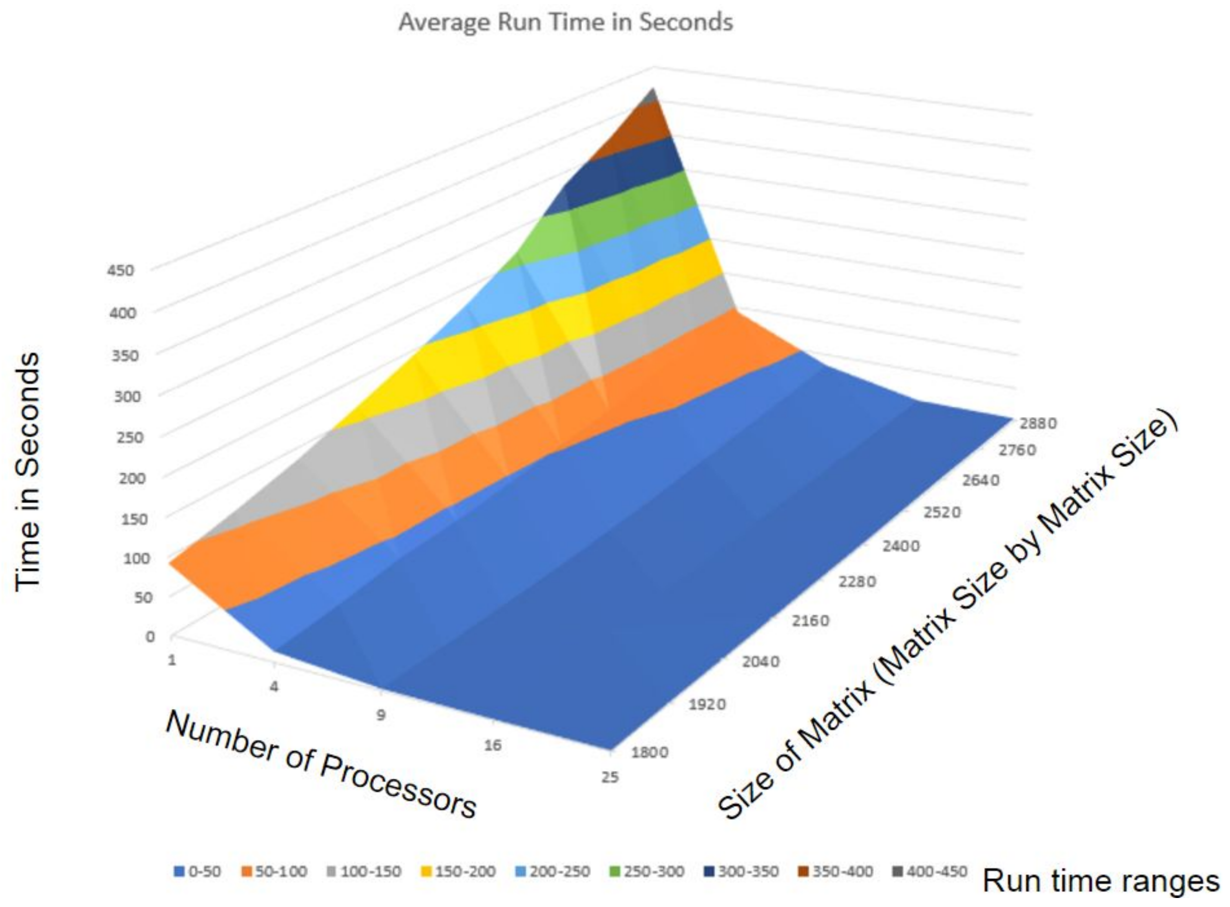
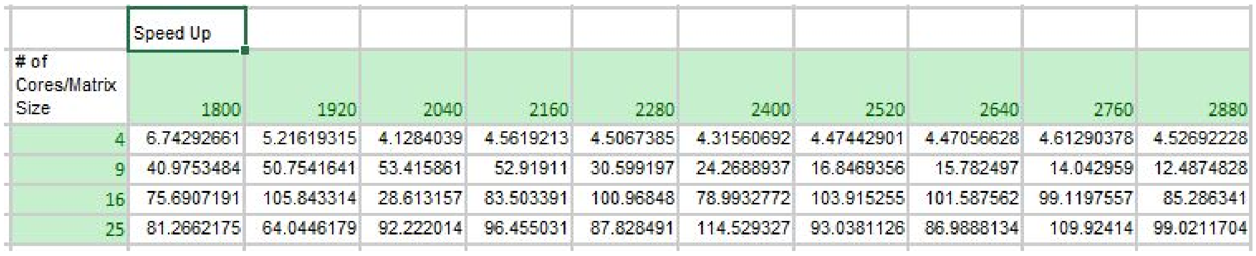


Figure 4: 3D graph that represent the runtimes based on the size of matrix and number of cores used. As the size of the matrix increases the slower the runtime is.

Part 3: Speed Up Analysis

Table 3: Table that shows the speed up of different matrices on different number of processors when executing Matrix multiplication.



	Speed Up									
# of Cores/Matrix Size	1800	1920	2040	2160	2280	2400	2520	2640	2760	2880
4	6.74292661	5.21619315	4.1284039	4.5619213	4.5067385	4.31560692	4.47442901	4.47056628	4.61290378	4.52692228
9	40.9753484	50.7541641	53.415861	52.91911	30.599197	24.2688937	16.8469356	15.782497	14.042959	12.4874828
16	75.6907191	105.843314	28.613157	83.503391	100.96848	78.9932772	103.915255	101.587562	99.1197557	85.286341
25	81.2662175	64.0446179	92.222014	96.455031	87.828491	114.529327	93.0381126	86.9888134	109.92414	99.0211704

As seen above in Table 3 the speed up varies from 4 to 105 speed. The formula for speed up is (sequential run time / parallel run time). While many of these speed up values are superlinear the reason for this is for the size of the cache. To show what the cache size for a processor is I did a small test. The test was to do matrix multiplication on a single processor on small matrices and figure out where data swapping between main memory and cache occurs. I started with a matrix size of 600x600 and increased the size by 20 and recorded the results which are seen in Fig. 5. By looking at the Fig. 5 we can see a jump at size 700x700 matrix. This is because the total number of integers used in a 700x700 sized matrix is larger than what the cache can handle. Swapping occurs between cache and main memory whenever data that is going to be used is currently in main memory and cannot be accessed in the cache. This data is swapped from main memory to the cache so it can be worked on. The reason there is a jump, run time increase, at size 700x700 matrix is due to this swapping that occurred, that means the cache can handle around 700^2 integers at a time before relying on main memory to hold data.

With knowing this information, these large speed up values make sense since each processors has its own cache. Thus when splitting up matrixA and matrixB by the number of processors, all the calculation such as moving matrices and multiplying matrices are all on each processor's cache since the sub matrices are very small. are all on the cache which is very fast.

Below Fig. 6 is a 3D visualization of Table 3 above which shows the speed up based on different matrix sizes and number of cores. As there are more cores and larger matrices, speed up tends to be more.

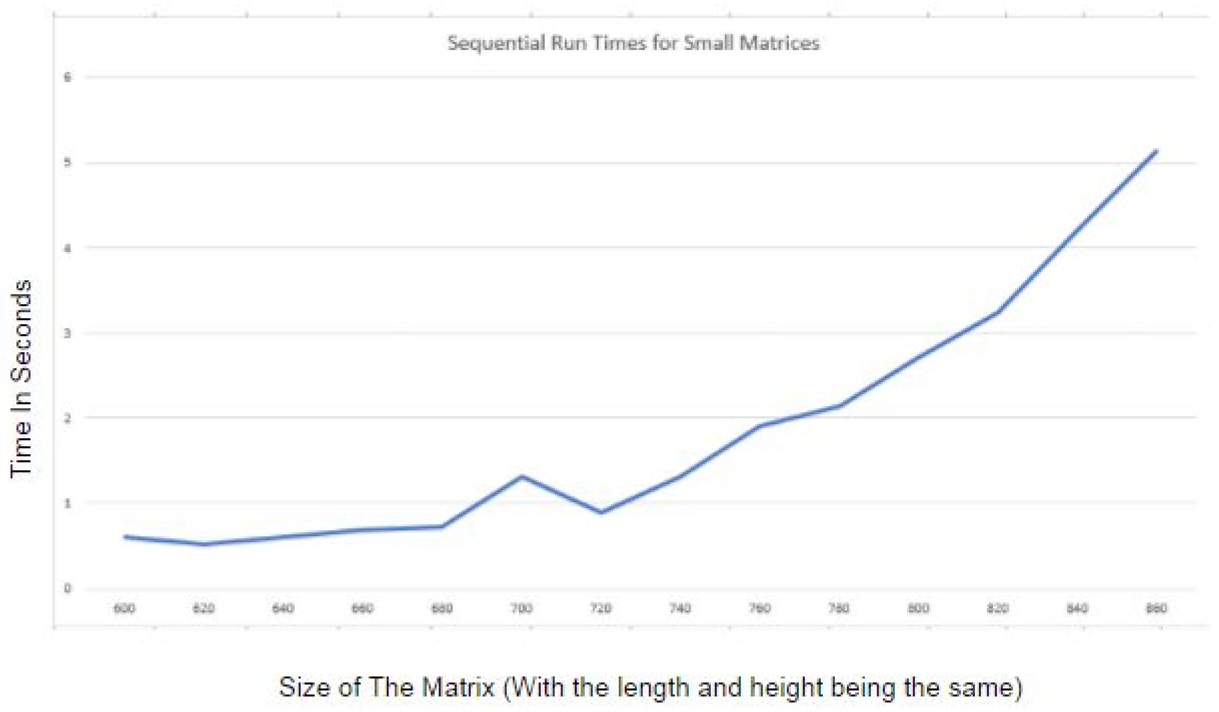


Figure 5: Sequential Matrix Multiplication Run Times on Smaller Matrices to figure out cache size. There is a “jump” at size 700x700 matrix size due to swapping between cache and main memory.

Table 4: Table that shows the efficiency of different matrices on different number of processors when executing Matrix Multiplication

	Efficiency									
# of Cores/Matrix Size	1800	1920	2040	2160	2280	2400	2520	2640	2760	2880
4	1.68573165	1.30404829	1.032101	1.1404803	1.1266846	1.07890173	1.11860725	1.11764157	1.15322595	1.13173057
9	4.55281649	5.63935156	5.9350956	5.8799011	3.3999108	2.69654374	1.87188173	1.75361077	1.56032878	1.38749809
16	4.73066994	6.61520714	1.7883223	5.2189619	6.31053	4.93707983	6.49470345	6.34922264	6.19498473	5.33039631
25	3.2506487	2.56178472	3.6888806	3.8582012	3.5131396	4.58117308	3.72152451	3.47955254	4.3969656	3.96084682

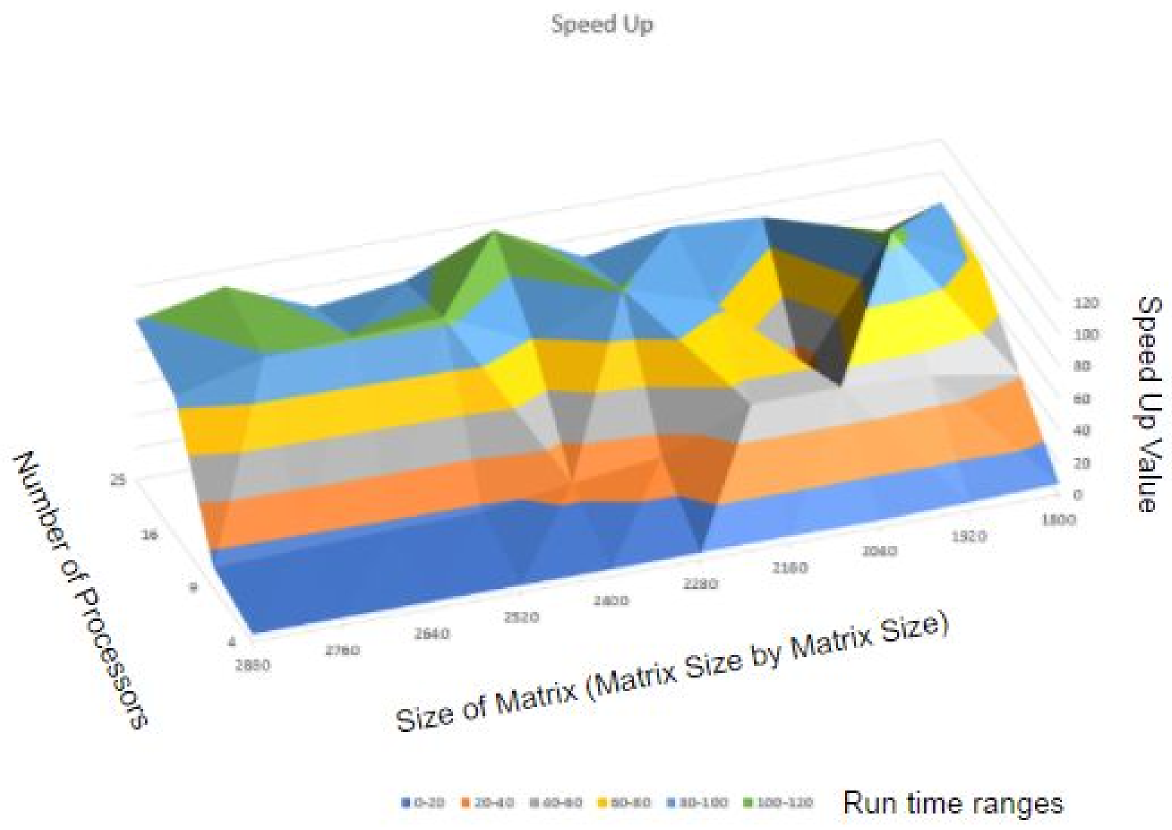


Figure 6: 3D graph that represent the speed up based on the size of matrix and number of Cores used. Generally, the speed up increases as there are more processors.

Part 4: Efficiency Analysis

Table 4 shows the efficiency of our multiple different types of tests. The equation for efficiency used is (speed up / number of processors). Generally as more processors are used the more efficient it is, however at times 16 cores can be more efficient that 25 cores. This could be true due to less message passing between processors.

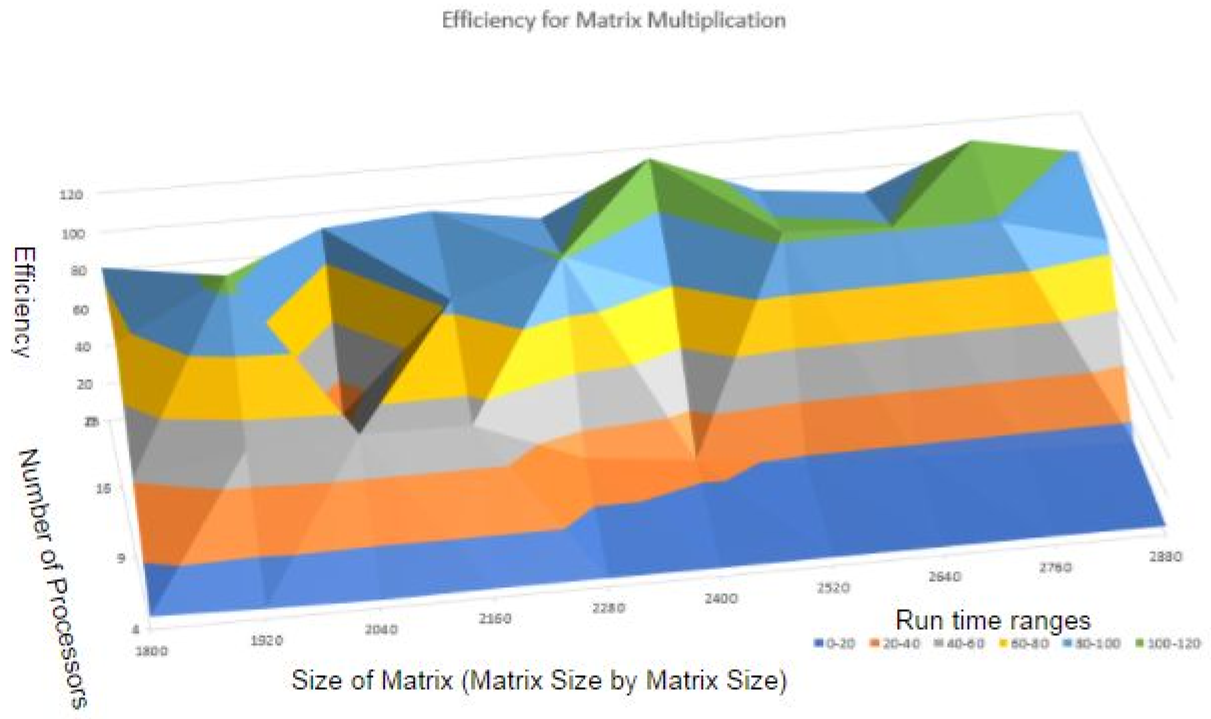


Figure 7: 3D graph that represent the efficiency based on the size of matrix and number of Cores used. Generally, the efficiency increases as there are more processors.

Problems with Parallel

Runtimes did not make sense until further analysis. It was difficult to mentally visualize how to do Cannon's algorithm. I had problems printing since I pass back all the result sub matrices back to the master.

Changes From Critics

One of the main comments all the commenters said about my code is that there is some repeated code for two of the functions, initRow and initCol. While they are exactly the same code with one line difference I did this initially to showcase that there are two moves one being a matrix move for rows and matrix move for columns. To fix this problem I did two things. First I made one move function (called move_MatrixRowOrCol) that can do both row and column matrix movement. To choose between where to move Rows and Columns there is an bool argument that is inputted in the function. Previously inside the initRow and initCol functions there was repeated code for the actual moving of the matricies. I changed to make the moves more modular by making a moveHelper function. This just does the same code as initRow did but separates it from the main move_MatrixRowOrCol function so it is more readable.

Another problem one of the commenters talked about is the moving data from a small sub matrix to a bigger final matrix or inputting data to a smaller sub matrix from a big matrix. I did not realize when writing my code that they were very similar so I made an inputArray function that can work for both types of cases. Also for readability I created a create2DMatrix function as some parts of the code need a buffer 2D matrix to work and that code to do this is reoccuring throughout the code.

Conclusion

This project teaches students about the Matrix Multiplication and how to parallelize the code. Due to the cache size there are many things that were learned/should be pointed out:

1) Cache size affects run time greatly.
2) Data in the Cache is much faster than in main memory
3) If there is more data than what the cache can handle, swapping between cache and main memory occurs thus slower run times.

Future Works

Students should try to parallelize this problem with 1D arrays, as it is much faster. If possible do this similar test with different types of supercomputers as the cache size can change, and number of cores may be much larger.