Jonathan Alexander Gibson
jagibson44, T00198998
Program 4 (MPI Matrix Multiplication)

During this programming assignment, I went through a few successes and setbacks, similar to that of the OpenMP matrix multiplication program. My first problem was a segmentation fault that took me a stunning 7~8 hours to debug. In the end, I noticed that one of my nested loop indexes was in the wrong place and was incrementing the row index where the column index should've been incremented.

Next, I couldn't seem to populate the matrices with random doubles, despite my successful implementation of this program with OpenMP and C++. Initially, I attempted to write the MPI program with C. Although I modified the "**range_rand_double()**" function to work with C, I couldn't figure out why my program wouldn't run the way I expected it to. This led me to learning about debugging and compiling MPI programs with GDB.

Through a lengthy process of trial and error, I eventually found the correct Bash commands to debug and compile the C++ MPI program with additional command line arguments. These Bash commands are outlined in "**Makefile**". After all this trouble, I was finally ready to start the main part of the assignment.

I went through a similar process for this MPI implementation as I did with the OpenMP implementation. I also attempted to make the programs as similar as possible (non-optimized, no dynamic matrix allocation, etc...) to not skew the execution time results in either direction (minimize implementation bias).

For a while, I pondered the way that I would partition the original matrices (A & B) to delegate the work amongst the multiple MPI processes. I considered sending the original matrices row by row (like dynamic partitioning), chunk by chunk (like static partitioning), and simply sending both entire matrices to each process to work on specific sections; with this last idea, my plan was to create rectangular, temporary matrices of unique sizes based on the number of processes being used, then to piece all of the temporary matrices back together after returning them to process 0. I found that these implementations were either too tedious or too complex for the scope of this program, and decided to go with a different method.

As it turns out, if you declare a variable, within an MPI program, above the "**MPI_Init()**" function, that variable can be shared amongst all the processes. I used this feature of MPI to my advantage and avoided using "**MPI_Send()**" or "**MPI_Recv()**" entirely. While statically declaring the resulting matrix (C) at the top of "**main()**", I declared A and B farther down in the code so that each process had its own instance of these randomized matrices (creating numerous instances of matrices A and B, effectively, has almost no effect on the execution time, as each process runs at a very similar speed to its neighbors). After these essential steps were completed, I simply implemented the same matrix multiplication algorithm that I used with the OpenMP implementation.

As I started running my code (using just "**Makefile**") and getting proper output (printing to the screen and outputting to a temporary file), I noticed that larger computations would run too long and allow the connection to the HPC server to time out before they completed. I resolved this issue by submitting batch jobs to the HPC server rather than hoping that the program would finish in time.

The results for my MPI matrix multiplication program are shown below.

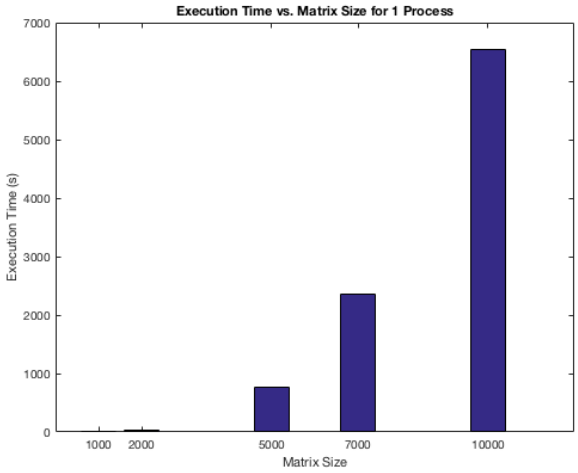

Figure 1: *Execution time for 1 process.*

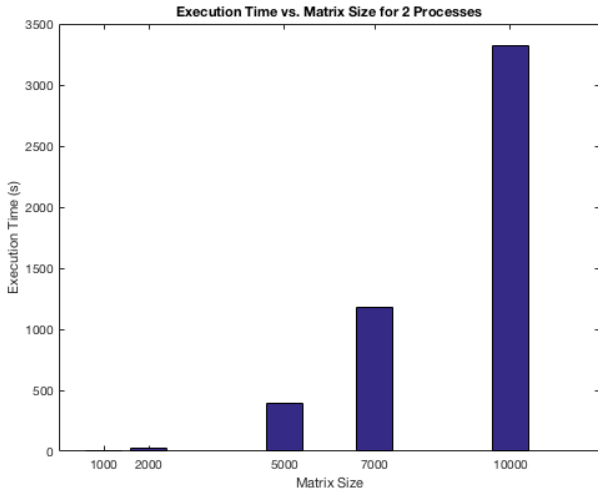

Figure 2: *Execution time for 2 processes.*

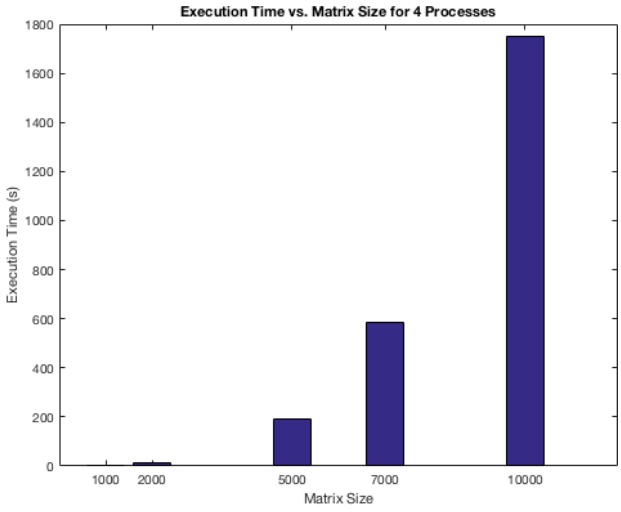

Figure 3: *Execution time for 4 processes.*

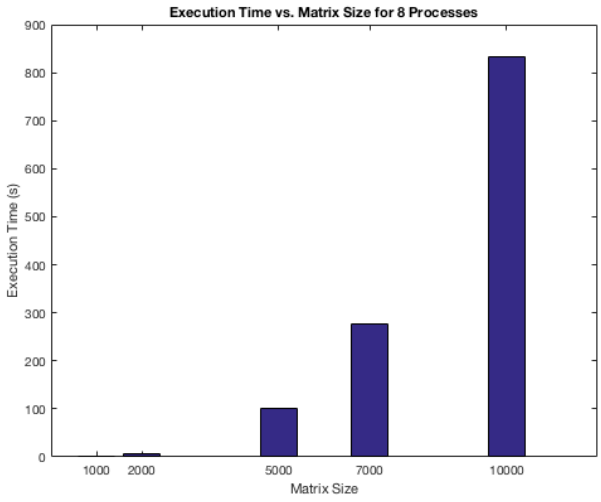

Figure 4: *Execution time for 8 processes.*

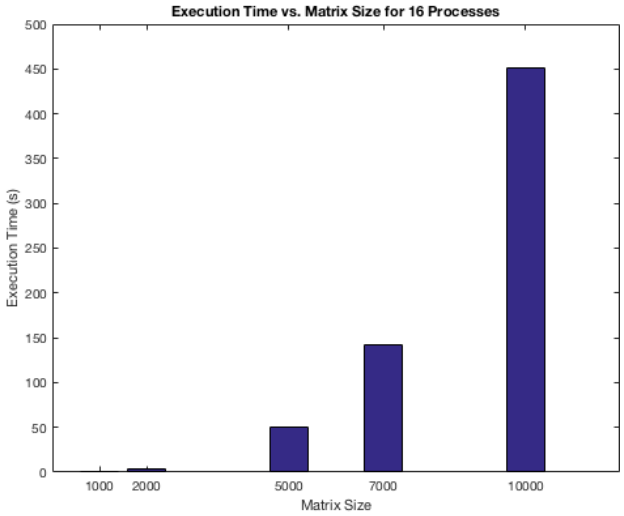

Figure 5: *Execution time for 16 processes.*

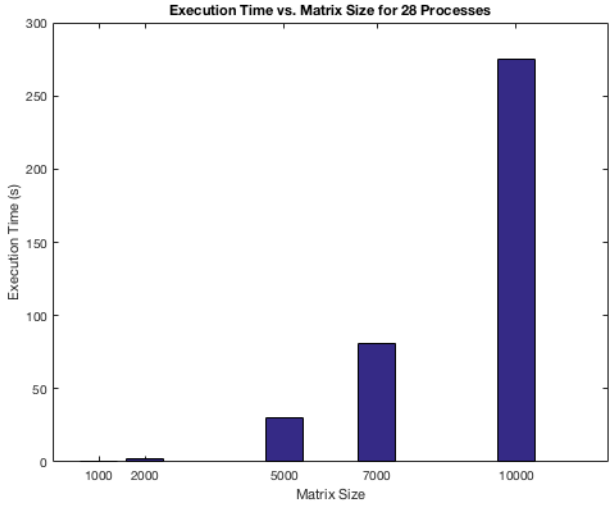

Figure 6: *Execution time for 28 processes.*

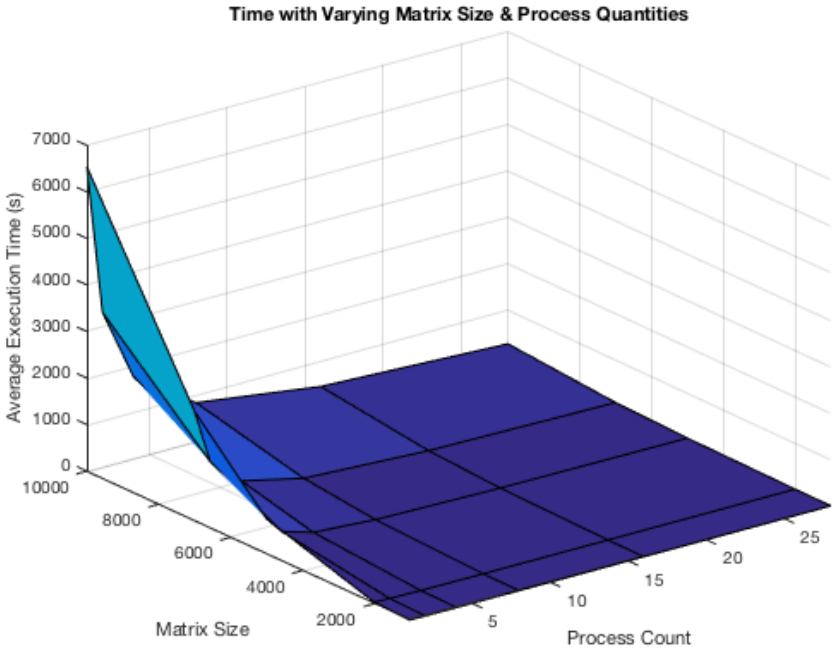

Figure 7: *MPI execution time surf plot based on varying matrix size and process count.*

Figure 1 through Figure 6 show scaled representations of the execution times for each process count, based on varying matrix sizes. These bar graphs all look very similar because they are scaled to the maximum execution time values for each set. I concatenated the data into a three-dimensional graph to get a more relative representation of the data. Figure 7 shows the gradient curve of the execution time for the matrix multiplications based on varying matrix size and process count. It is very clear that execution time drastically decreases with decreasing matrix size and increasing process count.

Execution Time on HPC Server

Matrix Size:	*Processes:*	*Time 1 (s):*	*Time 2 (s):*	*Time AVG (s):*	*Each Speedup:*
1000	1	4.744	4.733	4.739	1.000
1000	2	2.728	2.737	2.733	1.734
1000	4	1.376	1.365	1.371	3.457
1000	8	0.725	0.714	0.720	6.586
1000	16	0.359	0.321	0.340	13.937
1000	28	0.210	0.209	0.210	22.618
2000	1	35.691	34.728	35.210	1.000
2000	2	22.938	22.758	22.848	1.541
2000	4	11.307	11.229	11.268	3.125
2000	8	5.865	5.926	5.896	5.972
2000	16	3.031	2.955	2.993	11.764
2000	28	1.773	1.813	1.793	19.637
5000	1	758.351	758.601	758.476	1.000
5000	2	389.814	389.768	389.791	1.946
5000	4	193.890	192.405	193.148	3.927
5000	8	100.119	99.785	99.952	7.588
5000	16	51.052	50.874	50.963	14.883
5000	28	29.161	30.712	29.937	25.336
7000	1	2350.030	2353.820	2351.925	1.000
7000	2	1163.220	1201.150	1182.185	1.989
7000	4	587.658	583.878	585.768	4.015
7000	8	276.821	276.909	276.865	8.495
7000	16	141.617	142.836	142.227	16.536
7000	28	83.039	79.379	81.209	28.961
10000	1	6456.810	6623.350	6540.080	1.000
10000	2	3331.560	3311.360	3321.460	1.969
10000	4	1755.030	1742.180	1748.605	3.740
10000	8	832.166	833.838	833.002	7.851
10000	16	449.943	452.403	451.173	14.496
10000	28	241.107	308.458	274.783	23.801

Average Speedup Per Process Count:	1	1.000
	2	1.836
	4	3.653
	8	7.299
	16	14.323
	28	24.071

Table 1: *Collected Data.*

Using the average execution time, of two runs per process count, I calculated the speedup for each process count (above 1 process), with a constant matrix size. Then, I averaged these speedup times to get an approximate speedup curve, when using additional processes. Table 1 shows the extent of the data collected for the MPI matrix multiplication implementation.

Because the OpenMP matrix multiplication implementation turned out to provide super-linear speedups, I expected the MPI implementation to provide the same or similar speedup results. In the end, my expectations were mostly correct. Figure 8 shows that the MPI implementation was barely sub-linear (very close to achieving the ideal speedup curve). Like before, this near ideal speedup could be attributed to the cache effect and the lack of data dependency that matrix multiplication problems innately possess.

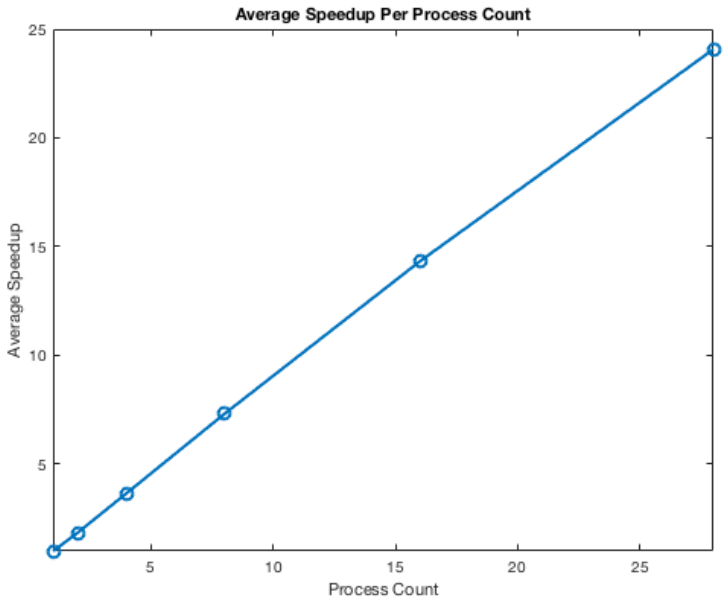

Figure 8: *Average MPI Speedup.*

The data, for the OpenMP and the MPI implementations, is fairly similar, especially with low matrix sizes. However, with larger matrix sizes and fewer threads/processes, the OpenMP implementation starts to diverge and shows significant execution time increases over the MPI implementation. As an example, matrices of size 10,000 were multiplied in an average of 12,619.750 seconds with 1 thread via OpenMP, where MPI computed the same result with 1 process in only 6,540.080 seconds, on average. In this case, the MPI implementation is almost twice as fast as the OpenMP implementation.

In hindsight, the OpenMP implementation could have achieved its large super-linear speedup due to the large execution time gap between 1 thread and multiple threads. The difference in time seems rather disproportionate, where the MPI data seems much more uniform with changes in process count.

In any case, according to the data, MPI seems like a better choice when confronted with programming problems that have potential for high optimization and low data dependency. The data does not correlate, however, OpenMP or MPIs abilities to deal with parallel overhead. A different problem set would need to be compared and analyzed to get an idea about that statistic. Although, I would expect that MPI may out perform OpenMP, once again, but not without additional programming cost.

In my opinion, OpenMP seems like a much more intuitive parallel programming library for those who don't want to deal with the tedious efforts of constantly sending individual messages between MPI processes (ultimately having to keep up with where data is located at all times). And although OpenMP tends to make assumptions for the programmer, it seems to do a good job at making use of its available resources to achieve sufficient performance increases. It seems that decreasing MPI overhead would be closely related to the programmer's ability to use MPI to its full potential. In other words, MPI has a steeper learning curve, but could potentially outperform OpenMP for many different problem sets.

The MATLAB code used to generate the MPI graphs is shown below.

```
%JONATHAN ALEXANDER GIBSON
%CSC 6740
%PARALLEL DISTRIBUTED ALGORITHMS
%DR. GHAFOOR
%PROGRAM 4 (MPI MATRIX MULTIPLICATION)


clear %clear all saved variable values
clc %clear the command window
close all %close all figures
format long %long variable format
%--------------------------------------------------------------------------


%(i) Average Speedup Per Process Count
ySpeedup = [1, 1.836, 3.653, 7.299, 14.323, 24.071];
xSpeedup = [1, 2, 4, 8, 16, 28];
figure;
plot(xSpeedup, ySpeedup, 'o-', 'LineWidth', 2, 'MarkerSize', 8);
ylim([1, 25]);
xlim([1, 28]);
ylabel('Average Speedup');
xlabel('Process Count');
title('Average Speedup Per Process Count');
%--------------------------------------------------------------------------

%(ii) 3D Surf Plot
z = [4.74, 35.21, 758.476, 2351.925, 6540.080;
    2.73, 22.85, 389.79, 1182.18, 3321.46;
    1.37, 11.27, 193.15, 585.77, 1748.61;
    0.72, 5.896, 99.952, 276.86, 833.002;
    0.34, 2.993, 50.96, 142.23, 451.17;
    0.21, 1.79, 29.94, 81.21, 274.78];
y = [1000, 2000, 5000, 7000, 10000;
    1000, 2000, 5000, 7000, 10000;
    1000, 2000, 5000, 7000, 10000;
    1000, 2000, 5000, 7000, 10000;
    1000, 2000, 5000, 7000, 10000;
    1000, 2000, 5000, 7000, 10000];
```

```
x = [1, 1, 1, 1, 1;
    2, 2, 2, 2, 2;
    4, 4, 4, 4, 4;
    8, 8, 8, 8, 8;
    16, 16, 16, 16, 16;
    28, 28, 28, 28, 28];
figure;
surf(x, y, z);
ylim([1000, 10000]);
xlim([1, 28]);
zlabel('Average Execution Time (s)');
ylabel('Matrix Size');
xlabel('Process Count');
title('Time with Varying Matrix Size & Process Quantities');
%-------------------------------------------------------------------------


%(iii) 1 Process
y1 = [4.74, 35.21, 758.476, 2351.925, 6540.080];
x1 = [1000, 2000, 5000, 7000, 10000];
figure;
bar(x1, y1);
ylabel('Execution Time (s)');
xlabel('Matrix Size');
title('Execution Time vs. Matrix Size for 1 Process');
%-------------------------------------------------------------------------


%(iv) 2 Processes
y2 = [2.73, 22.85, 389.79, 1182.18, 3321.46];
x2 = [1000, 2000, 5000, 7000, 10000];
figure;
bar(x2, y2);
ylabel('Execution Time (s)');
xlabel('Matrix Size');
title('Execution Time vs. Matrix Size for 2 Processes');
%-------------------------------------------------------------------------


%(v) 4 Processes
y4 = [1.37, 11.27, 193.15, 585.77, 1748.61];
x4 = [1000, 2000, 5000, 7000, 10000];
figure;
bar(x4, y4);
ylabel('Execution Time (s)');
xlabel('Matrix Size');
title('Execution Time vs. Matrix Size for 4 Processes');
%-------------------------------------------------------------------------
```

```
%(vi) 8 Processes
y8 = [0.72, 5.896, 99.952, 276.86, 833.002];
x8 = [1000, 2000, 5000, 7000, 10000];
figure;
bar(x8, y8);
ylabel('Execution Time (s)');
xlabel('Matrix Size');
title('Execution Time vs. Matrix Size for 8 Processes');
%-------------------------------------------------------------------------


%(vii) 16 Processes
y16 = [0.34, 2.993, 50.96, 142.23, 451.17];
x16 = [1000, 2000, 5000, 7000, 10000];
figure;
bar(x16, y16);
ylabel('Execution Time (s)');
xlabel('Matrix Size');
title('Execution Time vs. Matrix Size for 16 Processes');
%-------------------------------------------------------------------------


%(viii) 28 Processes
y28 = [0.21, 1.79, 29.94, 81.21, 274.78];
x28 = [1000, 2000, 5000, 7000, 10000];
figure;
bar(x28, y28);
ylabel('Execution Time (s)');
xlabel('Matrix Size');
title('Execution Time vs. Matrix Size for 28 Processes');
%-------------------------------------------------------------------------
```