Jonathan Alexander Gibson
jagibson44, T00198998
Program 2 (Matrix Multiplication)

During this programming assignment I went through several steps of success and setbacks. I started by writing a basic C++ program that would populate a matrix, with 16 (4x4) hardcoded values, and print it to the screen. After I got that working, I looked up a square matrix multiplication algorithm on Stack Overflow, familiarized myself with what was going on, and generated a correct matrix multiplication with the known values (I used Maple to confirm the result of the multiplication).

Next, I went back to some code that I got from Dr. Brown (from CSC 2100), to generate random values for the matrices, instead of hardcoded ones. Throughout this time, I also generated and deleted the matrices via dynamic allocation. But, when I couldn't figure out how to generate random ***double*** values for the dynamically allocated matrices, I simply reverted to static allocation.

After I confirmed that my code was working with small randomized matrices, I decided to test auto partitioning parallelism using ***#pragma omp parallel for*** on slightly larger matrices. I used the HPC server to test the automatic partitioning and confirm that the values were still correct. At this point, I was a bit stuck. I wasn't sure how to manually partition the work, so I consulted Steven and Ahsan for insight. Steven showed me a rough draft of how he thought the auto partitioning should behave based on the ***matrix_size***, ***thread_ID***, and calculated ***chunk_size***, ***start***, and ***end*** values.

After modifying his notes to work with my code, I made some progress, but also ran into some problems. My calculated ***end*** value was "off by one", and my final matrix was only generating values down the diagonal of the matrix, equal to that of the current ***chunk_size***. I then drew out two matrices on some engineering paper, to walk myself through the steps, for troubleshooting. I soon figured out that my indexing scheme didn't match my calculations for ***end*** (using < instead of <=) and I had manually partitioned the work for both the columns and the rows. Once I fixed these two issues, my code worked as expected and generated correct results.

Although my code was functioning properly, I noticed that my execution times weren't changing much, despite the number of threads I was using. I generated nearly 80% of my output files before I realized that I needed to increase the ***–cpus-per-task*** sbatch value to see any accurate speedup. Once I rectified this problem, my execution times started decreasing proportionally with the number of threads, and I started generating sound data, shown in Table 1. I then used the data to generate plots with MATLAB, shown below.

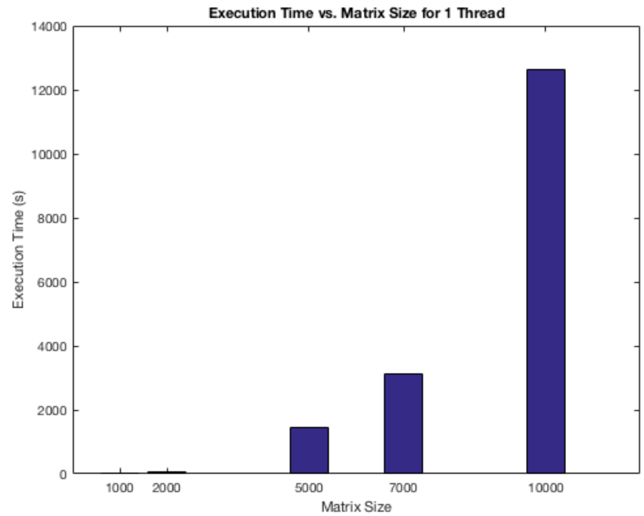

Figure 1: *Execution time for 1 thread.*

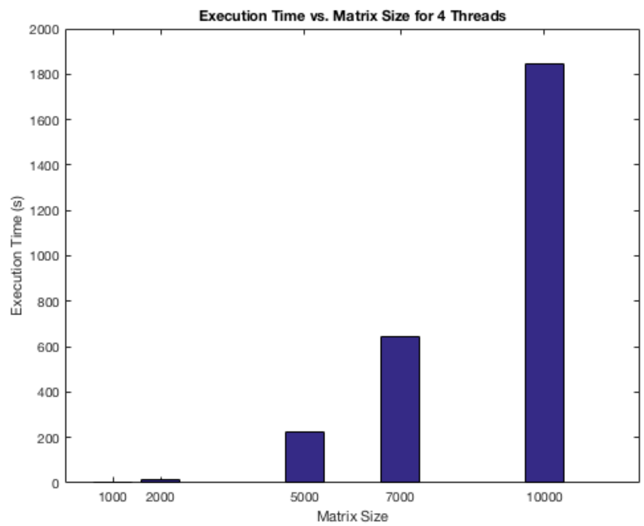

Figure 2: *Execution time for 4 threads.*

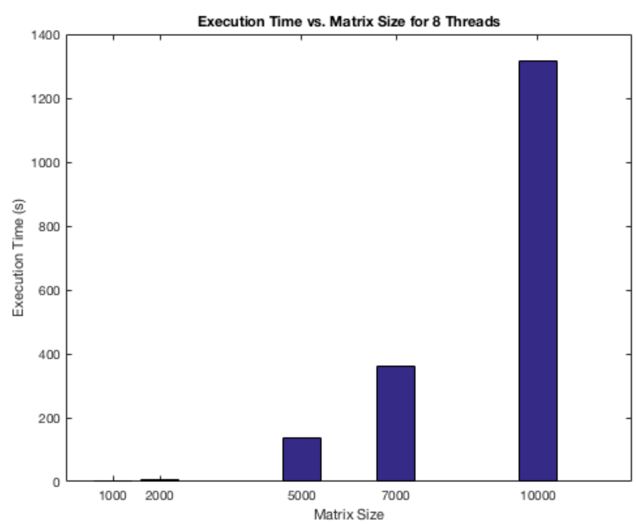

Figure 3: *Execution time for 8 threads.*

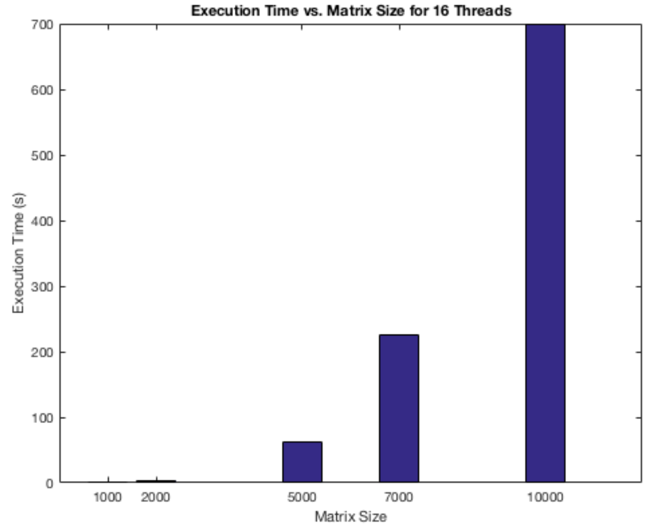

Figure 4: *Execution time for 16 threads.*

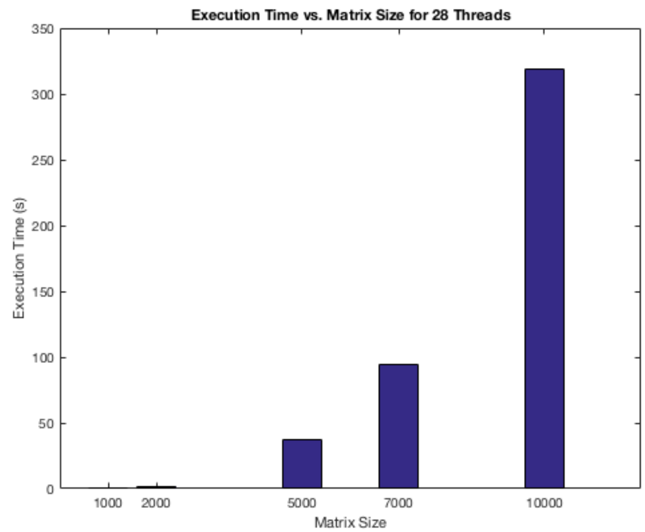

Figure 5: *Execution time for 28 threads.*

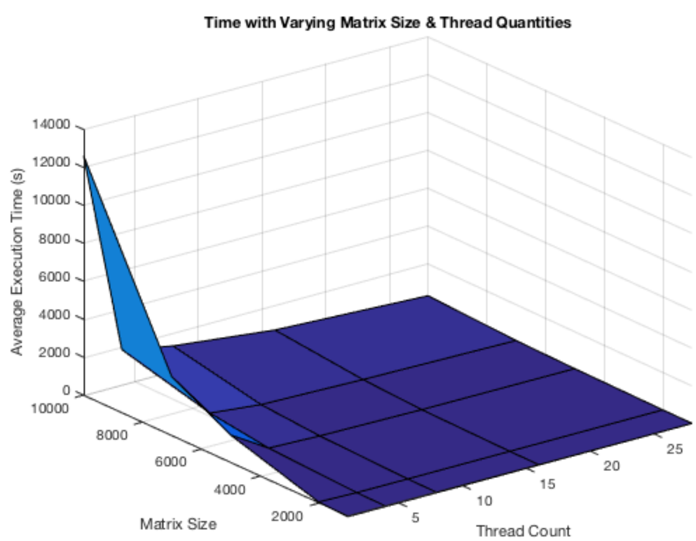

Figure 6: *Time surf plot based on varying matrix size and thread count.*

Figure 1 through Figure 5 show scaled representations of the execution times for each thread count, based on varying matrix sizes. These bar graphs all looks 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 6 shows the gradient curve of the execution time for the matrix multiplications based on varying matrix size and thread count. It is very clear that execution time drastically decreases with decreases in matrix size and increases in thread count.

Execution Time on Mac

Matrix Size:	*Threads:*	*Time (s):*
500	1	1
1000	1	19
1500	1	41
2000	1	165
2500	1	292
3000	1	557
3500	1	1174
4000	1	2528
4500	1	4374
5000	1	6498

Execution Time on HPC Server

Matrix Size:	*Threads:*	*Time 1 (s):*	*Time 2 (s):*	*Time AVG (s):*	*Each Speedup:*
1000	1	5.482	5.043	5.262	1.000
1000	4	1.055	1.040	1.048	5.023
1000	8	0.565	0.563	0.564	9.324
1000	16	0.456	0.463	0.460	11.445
1000	28	0.251	0.280	0.266	19.806
2000	1	67.940	67.988	67.964	1.000
2000	4	13.575	13.342	13.458	5.050
2000	8	5.438	5.200	5.319	12.778
2000	16	3.923	3.889	3.906	17.399
2000	28	1.987	1.842	1.914	35.505
5000	1	1464.360	1470.400	1467.380	1.000
5000	4	227.395	221.024	224.210	6.545
5000	8	112.019	161.922	136.971	10.713
5000	16	61.497	61.501	61.499	23.860
5000	28	37.475	37.407	37.441	39.192
7000	1	3773.740	2454.950	3114.345	1.000
7000	4	661.906	621.378	641.642	4.854
7000	8	287.240	435.322	361.281	8.620
7000	16	219.789	230.229	225.009	13.841
7000	28	94.199	94.671	94.435	32.979
10000	1	12420.500	12819.000	12619.750	1.000
10000	4	1870.840	1823.000	1846.920	6.833
10000	8	1125.580	1507.100	1316.340	9.587
10000	16	599.075	796.837	697.956	18.081
10000	28	304.413	333.028	318.721	39.595

Average	1	1.000
Speedup	4	5.661
Per	8	10.205
Thread	16	16.925
Count:	28	33.415

Table 1: *Collected Data.*

Using the average execution time, of two runs per thread count, I calculated the speedup for each thread count, with constant matrix size. Then, I averaged these speedup times to get an approximate speedup curve, when using additional threads.

To my surprise, the speedup curve shows that the speedup is super-linear. Although this is non-intuitive, there are a few reasons why this might be the case. First, matrix multiplication has no data dependency; none of the threads need to wait on each other for any calculations. This significantly reduces both overhead and execution time compared to dependent data sets. Second, the code was operating on simple, low-level computations; the complexity did not increase as the size of the matrices increased. Finally, the executions could have taken advantage of the cache effect; with an increase in CPU resources, there is also an increase in the amount of available cache. This could potentially reduce the data retrieval time from memory, by storing larger amounts of data in cache, instead of RAM, with the increasing number of CPUs. The speedup results are shown in Figure 7 below.

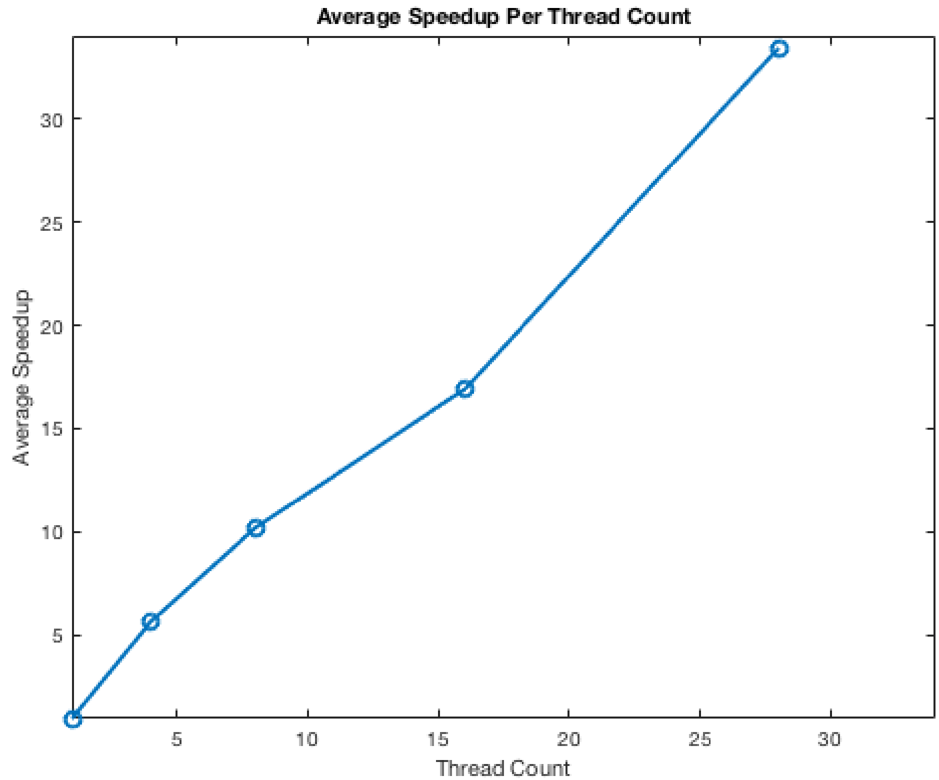

Figure 7: *Average Speedup.*

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

```
%JONATHAN ALEXANDER GIBSON
%CSC 6740
%PARALLEL DISTRIBUTED ALGORITHMS
%DR. GHAFOOR
%PROGRAM 2 (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 Thread Count
ySpeedup = [1, 5.661, 10.204, 16.9, 33.4];
xSpeedup = [1, 4, 8, 16, 28];
figure;
plot(xSpeedup, ySpeedup, 'o-', 'LineWidth', 2, 'MarkerSize',8);
ylim([1, 34]);
xlim([1, 34]);
ylabel('Average Speedup');
xlabel('Thread Count');
title('Average Speedup Per Thread Count');
%---------------------------------------------------------------------------


%(ii) 3D Surf Plot
z = [5.26, 67.96, 1467.38, 3114.345, 12619.75;
    1.05, 13.46, 224.21, 641.642, 1846.92;
    0.564, 5.32, 136.97, 361.28, 1316.34;
    0.46, 3.91, 61.5, 225, 698;
    0.266, 1.91, 37.44, 94.43, 318.72];
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];
x = [1, 1, 1, 1, 1;
    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('Thread Count');
title('Time with Varying Matrix Size & Thread Quantities');
%-------------------------------------------------------------------------

%(iii)
y1 = [5.26, 67.96, 1467.38, 3114.345, 12619.75];
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 Thread');
%-------------------------------------------------------------------------


%(iv)
y4 = [1.05, 13.46, 224.21, 641.642, 1846.92];
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 Threads');
%-------------------------------------------------------------------------


%(v)
y8 = [0.564, 5.32, 136.97, 361.28, 1316.34];
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 Threads');
%-------------------------------------------------------------------------


%(vi)
y16 = [0.46, 3.91, 61.5, 225, 698];
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 Threads');
%-------------------------------------------------------------------------


%(vii)
y28 = [0.266, 1.91, 37.44, 94.43, 318.72];
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 Threads');
%-------------------------------------------------------------------------
```