Jake Wheeler
CS415
05/04/2017
PA4 - Matrix Multiplication

Overview

This project focuses on the multiplication of square matrices. The program will be implemented and timed in a sequential method as well as a parallelized method, and their times compared. An example of a square matrix multiplication can be seen below, in Figure 1.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Figure 1, Example of square matrix multiplication. (Source: StackExchange.com)

Sequential

The sequential implementation of matrix multiplication shows a superlinear execution time, that is, as the square size of the matrices increase, the execution time increases faster and faster. The execution time vs. time graph should indeed look very similar to a graph of n^2, because essentially, the sequential implementation is doing size^2 calculations. The execution time vs. size graph can be seen below, in Figure 2.

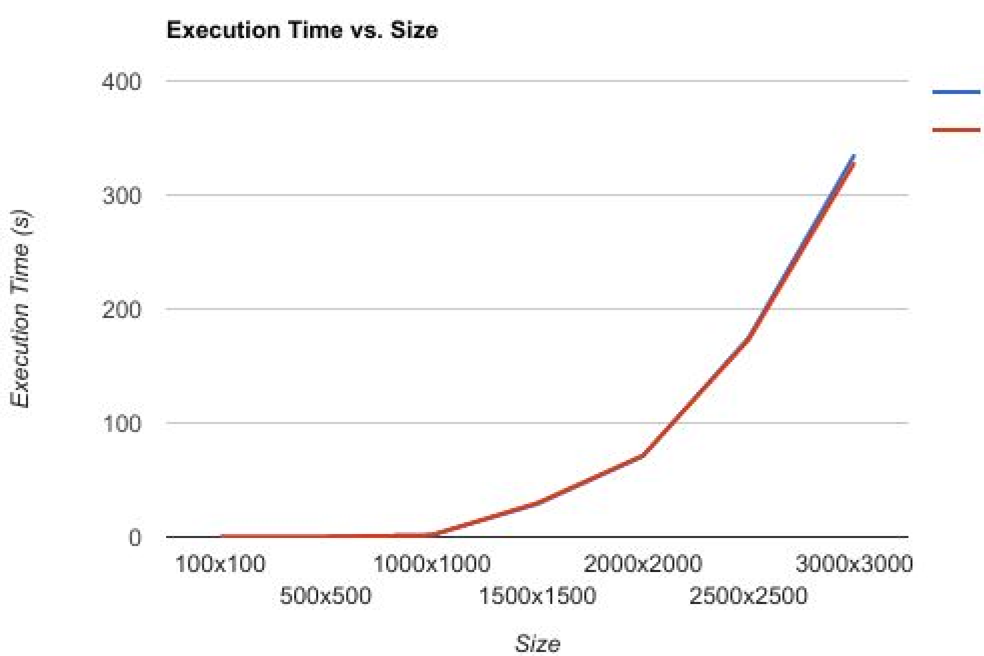

Figure 2: Figure 2, above, shows the relationship between two matrices of size <size>^2 being multiplied together, and the execution time in seconds..

	100x100	500x500	1000x1000	1500x1500	2000x2000	2500x2500	3000x3000
Time 1 (s)	0.001023	0.187337	1.59314	28.9215	70.8782	174.88	335.39
Time 2 (s)	0.001035	0.182211	1.59539	29.8491	71.5264	173.092	328.41

Table 1: Table 1, above, shows data for various tests logging the execution time of sequential matrix multiplication given different size square matrices.

Parallel

The parallel implementation uses Cannon's algorithm in order to shift the matrices around to achieve matrix multiplication. Through my results timing, I saw slowdown/insignificant speedup when running the parallel implementation on a smaller amount of cores (4 cores), and better speedup the more cores that were added. The runtime results can be seen below, in table 2 and figure 3.

	120	540	1020	1500	2040	2540	3000
1 Core	0.002099	0.395501	14.1909	49.5906	128.568	247.882	423.353
4 Cores	0.065275	1.85546	16.2948	38.4565	45.5429	81.97722	147.5589
9 Cores	0.045936	1.00796	5.04614	13.7304	42.634	68.2144	102.3216
16 Cores	0.031368	0.741969	3.31085	8.3644	39.5559	55.37826	94.143042

Table 2: Table 2, above, shows data for various tests logging the execution time of parallel matrix multiplication given different size square matrices and different numbers of cores.

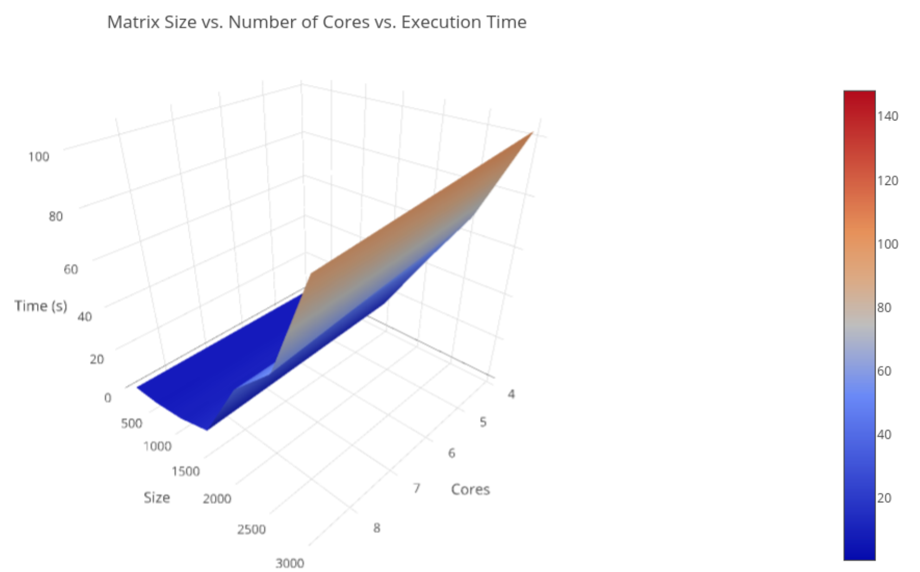

Figure 3: Figure 3, above, shows the relationship between two matrices of size <size>^2 being multiplied together via Cannon's algorithm, the number of parallelized cores, and the execution time in seconds via a 3D surface graph.

The execution time of the parallelized implementation of Cannon's algorithm does certainly show speedup, but not to the potential it could be. Other implementations of Cannon's algorithm show as much as a speedup factor of 200, which is much much more than my implementation. Upon further investigation, the bottleneck seems to be the implementation of my matrix_multiply function, which is called multiple times to multiply the separated and shifted matrices.

Table 3 and Figure 4 below show the speedup factor during the parallel implementation running on various cores (left column) and various sizes (top row).

	120	**540**	**1020**	**1500**	**2040**	**2540**	**3000**
4	0.032	0.213	0.871	1.290	2.823	3.024	2.869
9	0.046	0.392	2.812	3.612	3.016	3.634	4.137
16	0.067	0.533	4.286	5.929	3.250	4.476	4.497

Table 3: Table 3, above, shows the speedup factor (parallel/sequential) of the execution time when run on various numbers of cores.

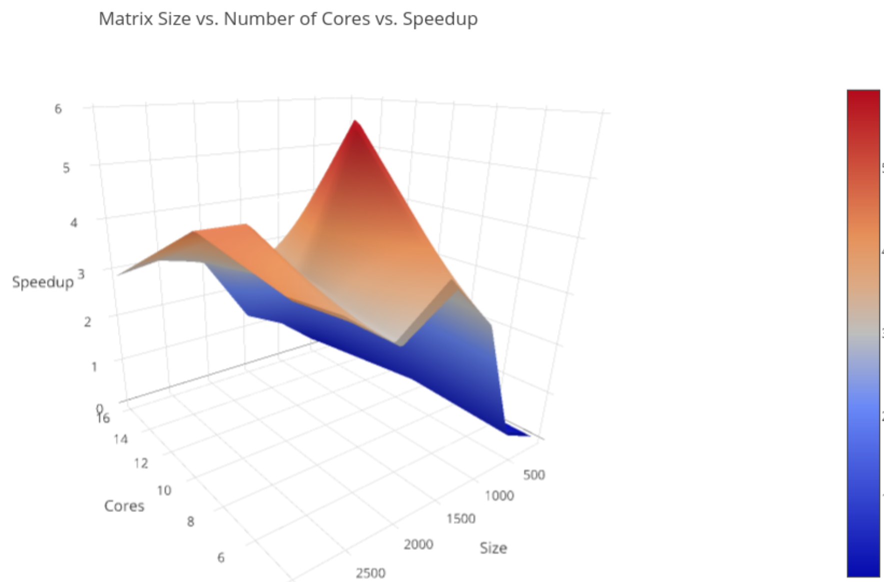

Figure 4: Figure 4, above, shows a 3D surface graph relating the speedup factor, the size of the matrix, and the number of cores.

The speedup data also shows questionable results. There are a few spikes, which could be attributed to various network variables. The low speedup is expected given the slow execution time, but speedup is still certainly present. The surface graph of the speedup overall trends upwards and to the left, increasing given the larger amount of cores and larger sizes of matrices. This is due to the overhead of the communication, on small matrices and/or a small number of cores the amount of communication is just too much in order for speedup to be achieved.

Conclusion

My implementation of Cannon's algorithm to perform matrix multiplication on two given matrices could certainly use some work, but it does show speedup and shows the potential that the algorithm has. Given more time, I could profile my implementation and find what exactly is taking so much time, optimize that and achieve the full potential of the parallel implementation, Cannon's algorithm.