# PA4 – Matrix Multiplication Documentation

Brian Conway

4 May 2017

CS415

# Matrix Multiplication

This sequential implementation of matrix multiplication involves taking the dot product of each row in matrix A and with each column of matrix B in order to calculate the values of each cell in matrix C.

This parallel implementation of matrix multiplication involves using Cannon's algorithm to perform matrix multiplication in parallel, involving the shifting of rows in A and the shifting of columns in B.

## Methodology - Sequential

The file sequential.cpp is responsible for generating random values for square matrices A and B based on the dimensions specified, matrix multiplying them, and outputting statistics about the timing as well as optionally writing the three matrices to a file.

The program has one mandatory command line parameter for the dimensions of the square matrix, and an optional parameter for if you want the matrices output to a file. If the second command line parameter is "y", then the results are output to the file "output.txt".

The program first calls generateNumbers(), a function which uses c++11 random devices to generate uniformly distributed values from 0 to 999,999 for all of the cells of matrices A and B. The generator is seeded appropriately so that this sequential version generates the same numbers as the eventual parallel one.

Afterwards, the program calls the matrixMult() function, which loops through each row of A and column of B, then loops through each element in that row/column combination to sum up the multiplication of all corresponding elements and store the result in the corresponding cell in C. For example, to calculate the cell in row 3, column 2 of C, it takes the dot product of row 3 of matrix A and column 2 of matrix B.

It times this using the Timer class, and it repeats this process the amount of times specified by the NUM_MEASUREMENTS constant. Afterwards, it calls the calcStatistics() function to get an average of the time taken to multiply the matrices and outputs it to the console.

Timing is done using the custom Timer class, which has start, stop, and resume functions similar to a stopwatch. Timer makes use of timevals and the gettimeofday() function in order to get an accurate reading of seconds and milliseconds at certain instants. When the elapsed time is to be given, the Timer takes the difference in seconds and milliseconds between when the timer was started and when the timer was stopped. It then combines the seconds and milliseconds reading into a double precision floating point variable.

Methodology - Parallel

The file parallel_file.cpp is responsible for loading values for square matrices A and B from input files, matrix multiplying them in parallel with Cannon's algorithm, and outputting statistics about the timing as well as optionally writing the three matrices to a file.

The program has two mandatory command line parameters for the file names of the square matrices A and B, and an optional parameter for if you want the matrices output to a file called "C.txt". If the second command line parameter is "y", then the results are output to the file "output.txt". By default, the scripts "par_4.sh", "par_9.sh", and "par_16.sh" pass the filenames "A.txt" and "B.txt", also specifying "y" to print matrix C to a file.

The program first uses various MPI functions to create a comm world that represents the processes in 2D cartesian coordinates. The way processes tend to be assigned for 4 processes is as follows: Process 0 at (0,0), process 1 at (0,1), process 2 at (1,0), process 3 at (1,1). For larger amounts of processes it follows the same pattern. This allows for easy calculations for the shifts in Cannon's algorithm later with the function MPI_Cart_shift().

This is where the program diverges between master nodes and other nodes, with master first calling fileInput (), a function which loads the main A and B matrices with the numbers from their respective files. Afterwards, the master process copies its own chunks of A and B from the main matrices.

Then, the master process calls the function sendChunksFromMaster() to distribute the remaining chunks to their respective processes. The other processes simply receive the matrix size and their chunks of A and B.

This is where the program converges again, and all processes begin to run the same code. A barrier is called to synchronize all processes, then everyone initializes their values of the C chunk to 0. The timer is started and then matrix multiplication begins. In the initialization step of Cannon's algorithm, row chunk x of matrix A is shifted x places left and column chunk y of matrix B is shifted y places up. The function MPI_Cart_shift() is used to calculate source and destination processes, those values are passed into MP_Send_recv_replace() in order to use one buffer for sending and receiving the chunk.

After this initialization process, the main loop of Cannon's algorithm begins. Matrix chunks A and B are multiplied with the result accumulated into C, and then rows of A are shifted one place left and rows of B are shifted one place up. This happens sqrt(numTasks) times, so 3 times for 9 processes and 4 times for 16 processes, etc. When this finishes, each process has a completed chunk of matrix C.

Afterwards, it calls the calcStatistics() function to get an average of the time taken to multiply the matrices and outputs it to the console.

If the last command-line argument specified the resulting matrix be output to a file, the function outputResults() is called. Here, the master process copies its own chunk of C into main matrix C and then receives all other chunks from the other processes, copying them into their corresponding part of matrix C. Then, the resulting matrix C is output to the file "C.txt" in the same format as the input files.

## Results – Sequential

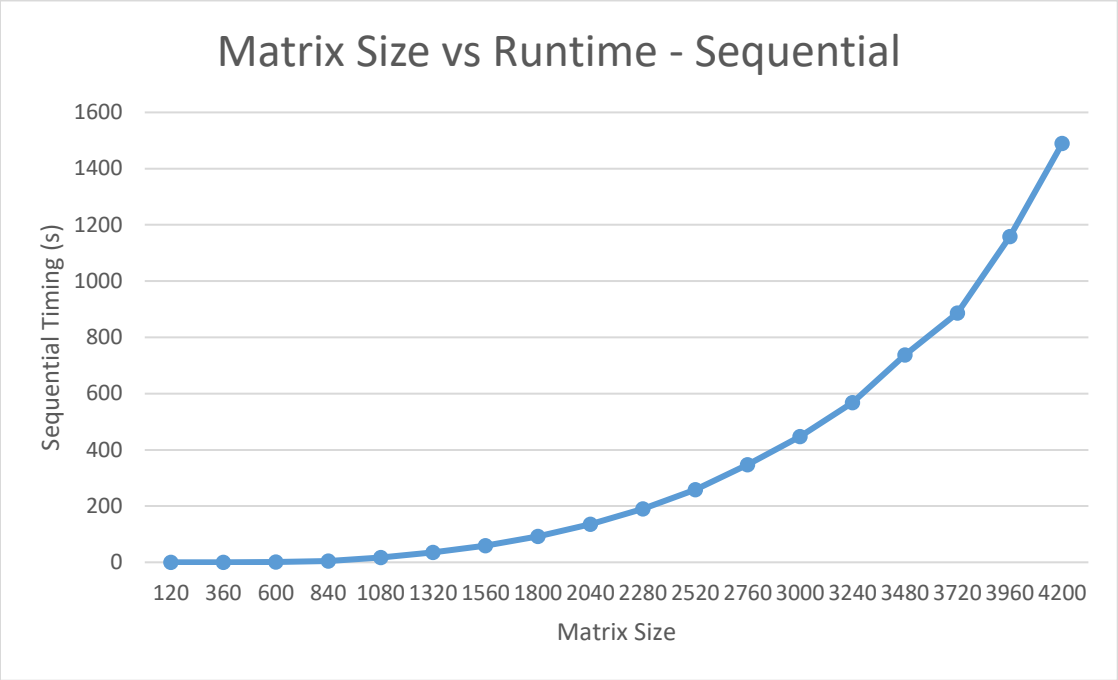


Fig. 1 Graph of sequential runtimes with different sizes of the matrices.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 |
|---|---|---|---|---|---|---|---|---|
| 0.006546 | 0.145585 | 0.716863 | 4.41264 | 17.2486 | 35.3099 | 58.9637 | 92.0221 | 135.221 |

Table 1: First half of timings for sequential matrix multiplication calculations, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 2280 | 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|---|
| 190.317 | 258.853 | 347.023 | 447.536 | 567.295 | 736.911 | 886.414 | 1158.14 | 1489.15 |

Table 2: Second half of timings for sequential matrix multiplication calculations, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

## Results – Sequential with Estimations

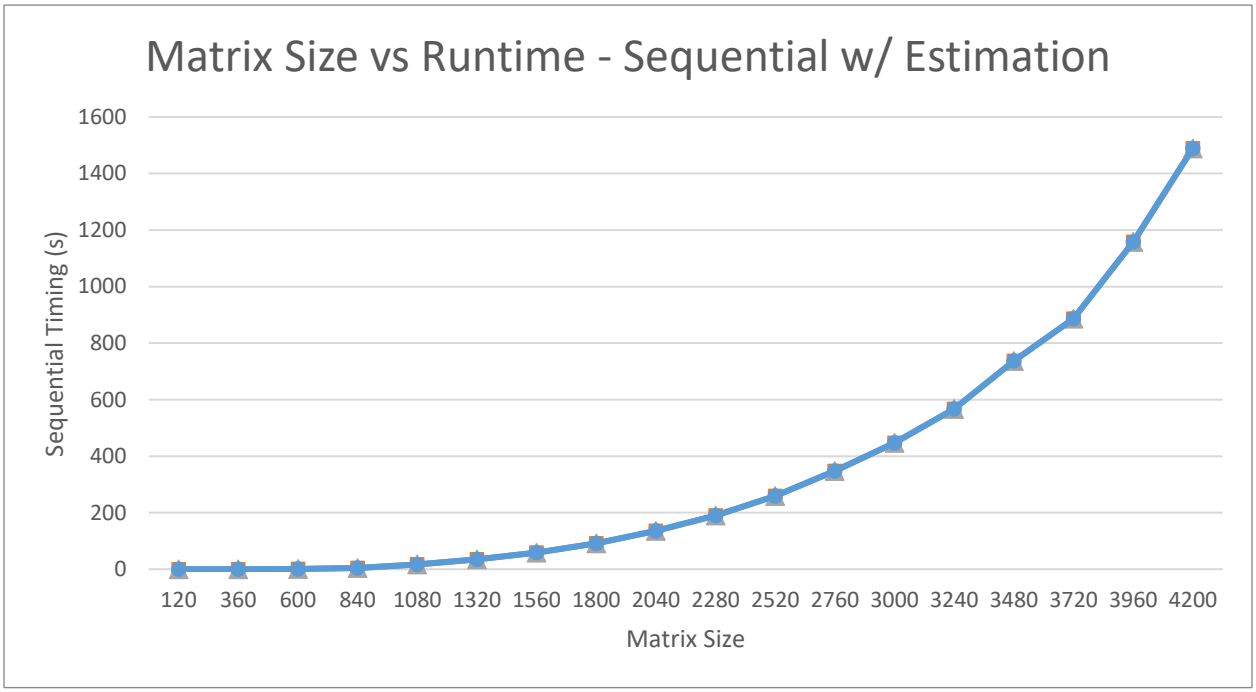


Fig. 2 Graph of sequential runtimes with estimated times for sizes beyond 4200, in order to compare to results for the parallel implementations.

| 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 | 6120 | 6360 |
|---|---|---|---|---|---|---|---|---|
| 78.8248 | 95.48 | 111.111 | 132.495 | 156.039 | 182.562 | 194.174 | 228.423 | 253.517 |

Table 3: First part of additional estimated timings for sequential matrix multiplication calculations, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 6600 | 6840 | 7080 | 7320 |
|---|---|---|---|
| 289.097 | 318.358 | 357.77 | 422.593 |

Table 4: Second half of additional estimated timings for sequential matrix multiplication calculations, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

# Results – Parallel

**Parallel – 4 process mesh**

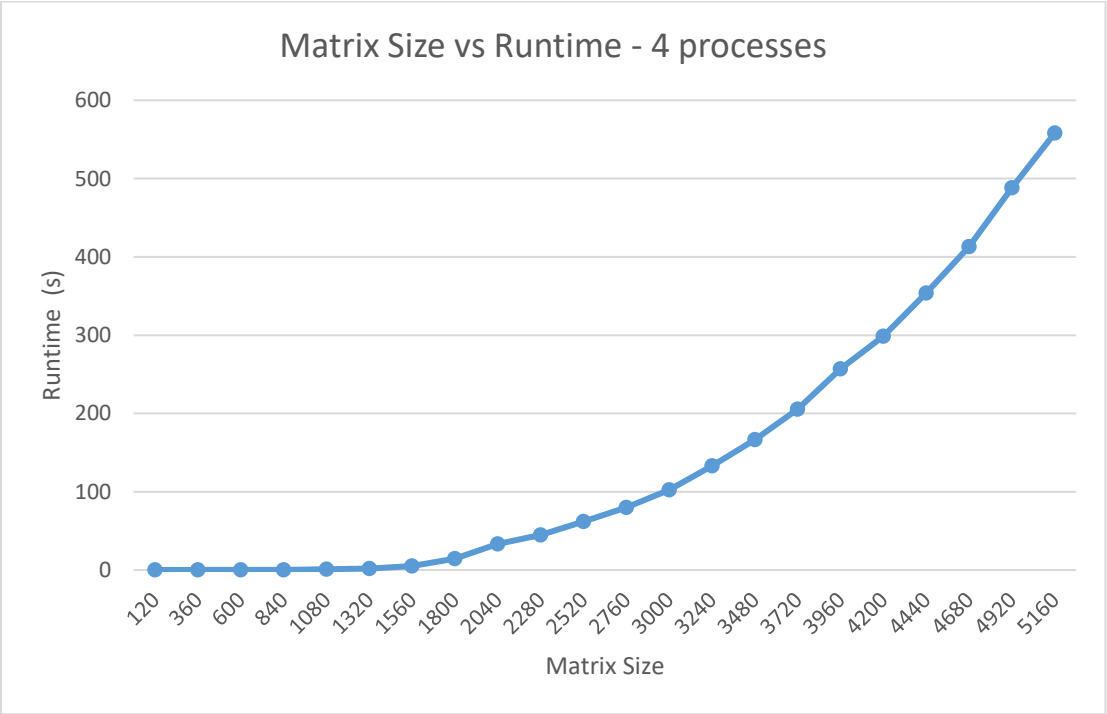


Fig. 3 Graph of runtimes with different sizes of the matrices for a 4 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 |
|---|---|---|---|---|---|---|---|---|
| 0.003484 | 0.021125 | 0.090451 | 0.293056 | 1.16535 | 1.99225 | 5.03024 | 14.4529 | 33.3656 |

Table 5: First part of timings for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 2280 | 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|---|
| 7.30272 | 16.048 | 27.1116 | 41.8183 | 56.834 | 72.3295 | 90.2638 | 112.372 | 131.245 |

Table 6: Second part of timings for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 4440 | 4680 | 4920 | 5160 |
|---|---|---|---|
| 353.809 | 413.03 | 488.305 | 558.123 |

Table 7: Last part of timings for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

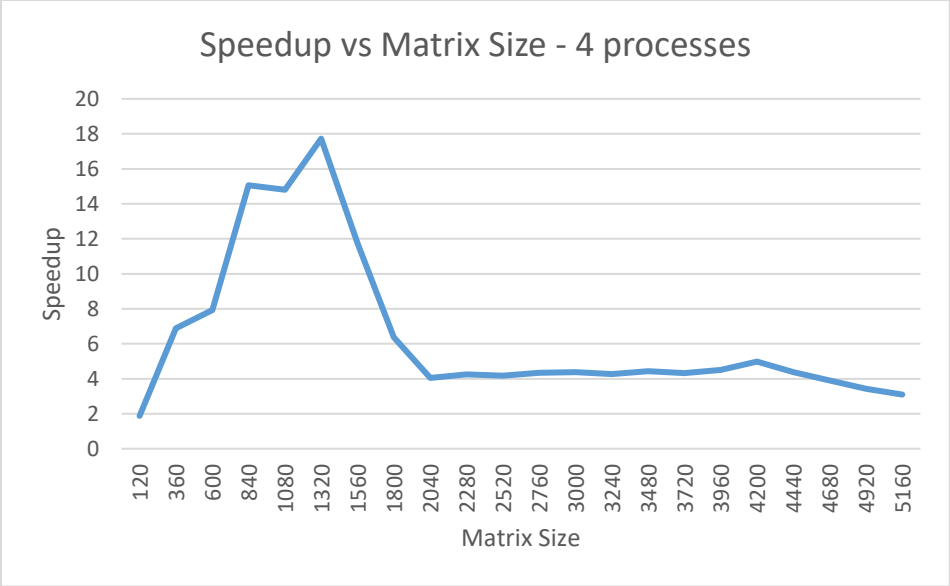


Fig. 4 Graph of speedup with different sizes of the matrices for a 4 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 |
|---|---|---|---|---|---|---|
| 1.88 | 6.89 | 7.93 | 15.06 | 14.80 | 17.72 | 11.72 |

Table 7: First part of speedup for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

| 1800 | 2040 | 2280 | 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|---|---|---|
| 6.367 | 4.0527 | 4.2572 | 4.182435 | 4.33798 | 4.3711 | 4.2583 | 4.4258 | 4.3195 | 4.5047 | 4.9873 |

Table 8: Second part of speedup for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

| 4440 | 4680 | 4920 | 5160 |
|---|---|---|---|
| 4.38 | 3.9 | 3.42 | 3.1 |

Table 9: Third part of speedup for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

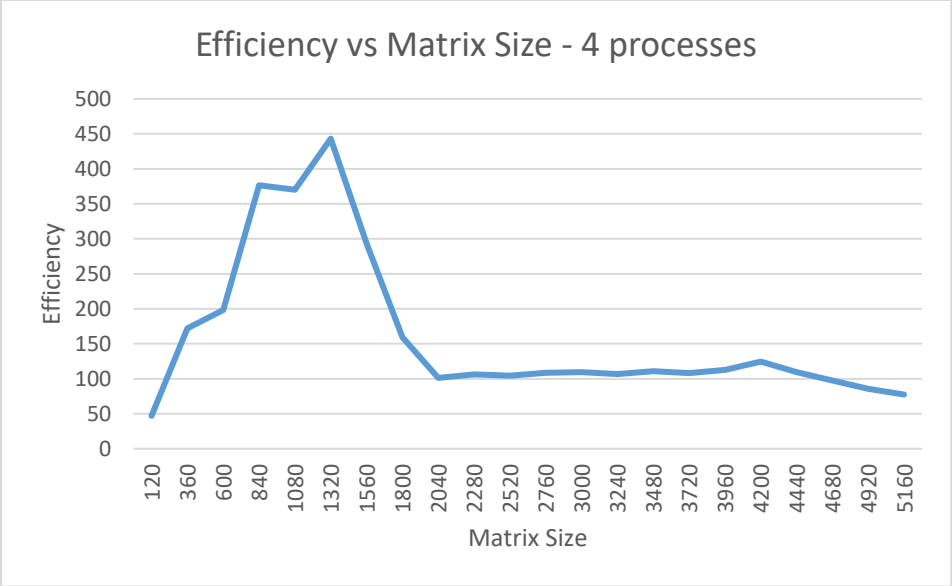


Fig. 5 Graph of efficiency with different sizes of the matrices for a 4 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 | 2280 |
|---|---|---|---|---|---|---|---|---|---|
| 46.97189 | 172.29 | 198.14 | 376.43 | 370.03 | 443.09 | 293.05 | 159.18 | 101.31 | 106.43 |

Table 10: First part of efficiency for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

| 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|
| 104.560 | 108.45 | 109.28 | 106.46 | 110.65 | 107.99 | 112.62 | 124.68 |

Table 11: Second part of efficiency for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

| 4440 | 4680 | 4920 | 5160 |
|---|---|---|---|
| 109.45 | 97.39 | 85.45 | 77.45 |

Table 12: Third part of efficiency for matrix multiplication calculations with 4 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

**Parallel – 9 process mesh**

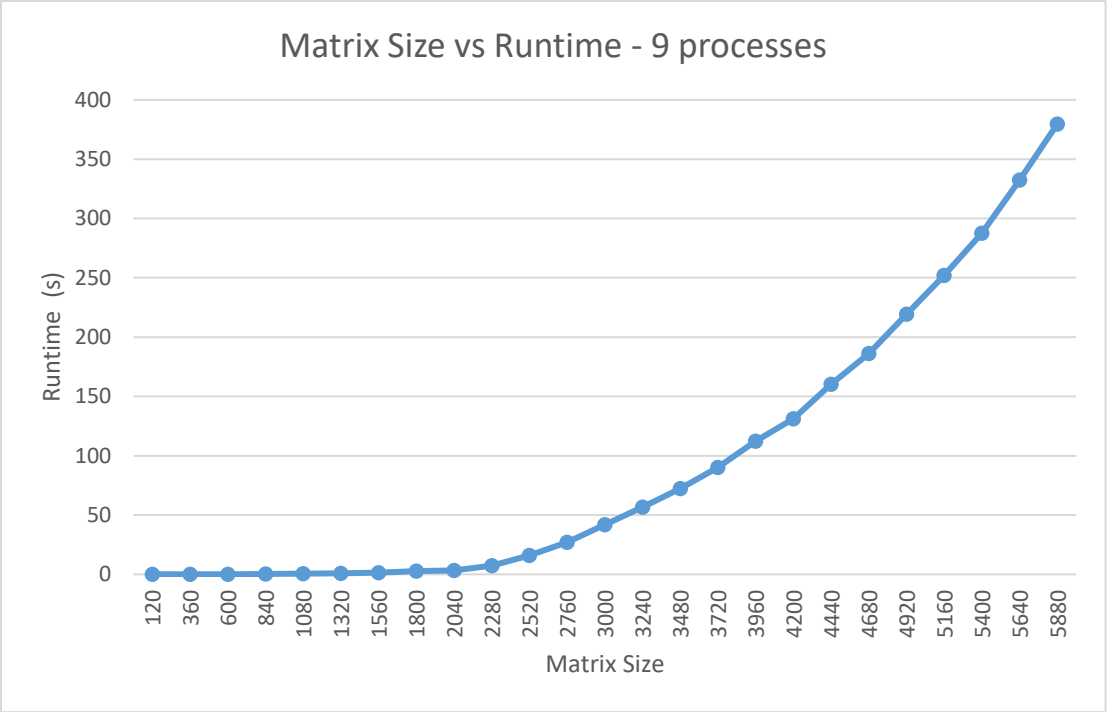


Fig. 6 Graph of runtimes with different sizes of the matrices for a 9 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 |
|---|---|---|---|---|---|---|---|---|
| 0.112949 | 0.125255 | 0.135314 | 0.313103 | 0.690703 | 0.93117 | 1.4113 | 2.76752 | 3.34771 |

Table 13: First third of timings matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 2280 | 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|---|
| 7.30272 | 16.048 | 27.1116 | 41.8183 | 56.834 | 72.3295 | 90.2638 | 112.372 | 131.245 |

Table 14: Second third of timings for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 |
|---|---|---|---|---|---|---|
| 160.431 | 186.143 | 219.277 | 251.983 | 287.508 | 332.462 | 379.619 |

Table 15: Last third of timings for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

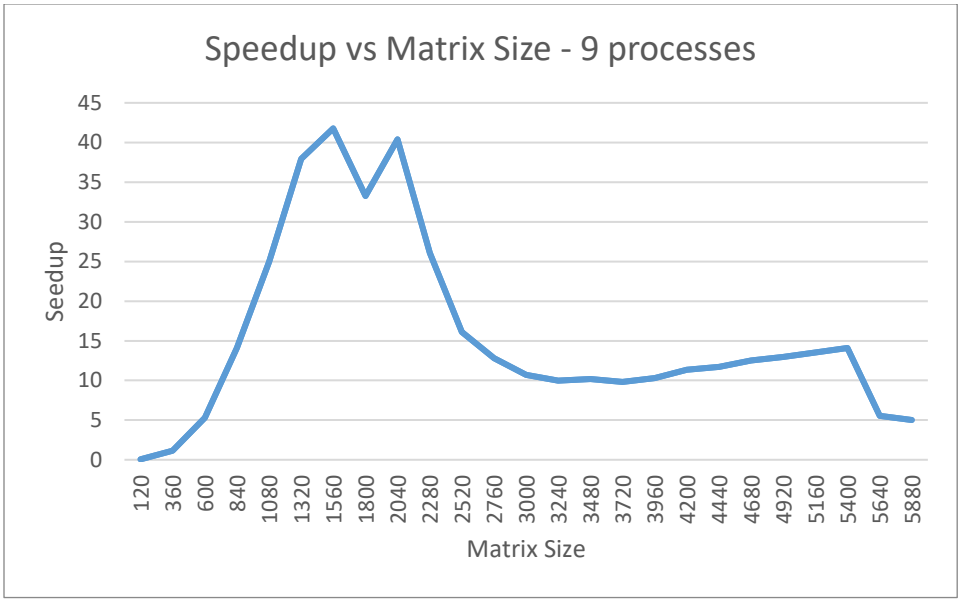


Fig. 7 Graph of speedup with different sizes of the matrices for a 9 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 |
|---|---|---|---|---|---|---|
| 0.058 | 1.162 | 5.298 | 14.093 | 24.973 | 37.92 | 41.78 |

Table 16: First part of speedup for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

| 1800 | 2040 | 2280 | 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|---|---|---|
| 33.251 | 40.392 | 26.061 | 16.13 | 12.8 | 10.702 | 9.982 | 10.188 | 9.82 | 10.306 | 11.346 |

Table 17: Second part of speedup for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

| 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 |
|---|---|---|---|---|---|---|
| 11.72 | 12.52 | 12.95 | 13.53 | 14.09 | 5.56 | 5.03 |

Table 18: Third part of speedup for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

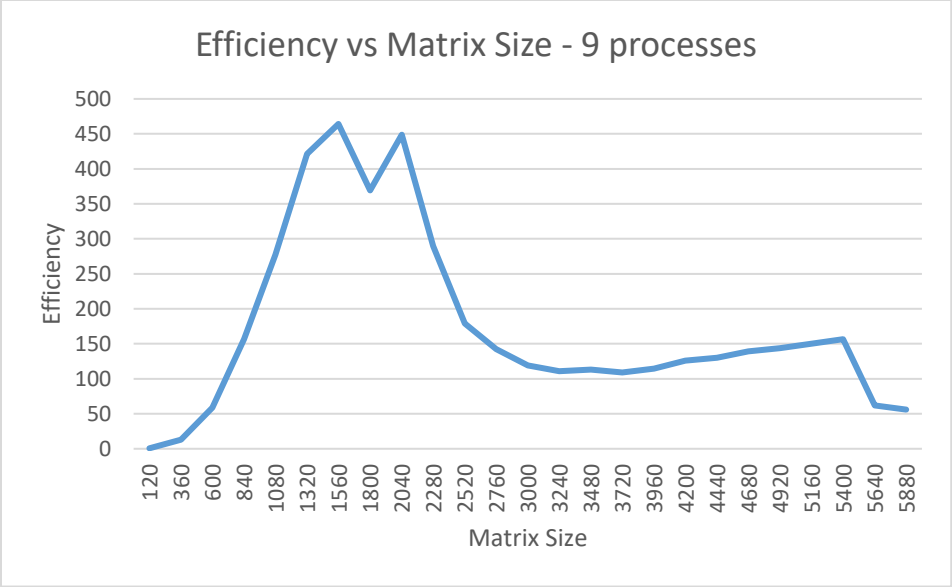


Fig. 8 Graph of efficiency with different sizes of the matrices for a 9 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 | 2280 | 2520 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.644 | 12.915 | 58.864 | 156.592 | 277.473 | 421.333 | 464.219 | 369.453 | 448.801 | 289.568 | 179.221 |

Table 19: First part of efficiency for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

| 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 | 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 142.22 | 118.91 | 110.91 | 113.2 | 109.11 | 114.51 | 126.07 | 130.2 | 139.08 | 143.91 | 150.36 | 156.52 | 61.79 | 55.87 |

Table 20: Second part of efficiency for matrix multiplication calculations with 9 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

**Parallel – 16 process mesh**

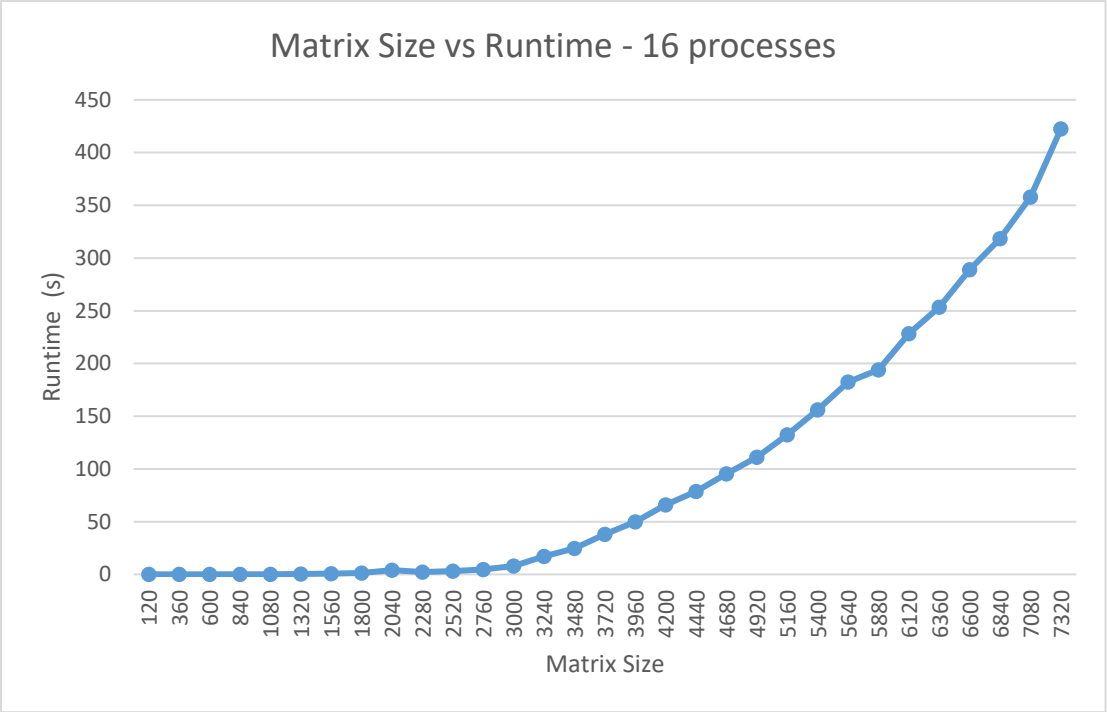


Fig. 9 Graph of runtimes with different sizes of the matrices for a 16 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 |
|---|---|---|---|---|---|---|---|---|
| 0.172028 | 0.048591 | 0.117823 | 0.153232 | 0.247421 | 0.491062 | 0.754605 | 1.44936 | 3.93138 |

Table 21: First part of timings matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 2280 | 2520 | 2760 | 3000 | 3240 | 3480 | 3720 | 3960 | 4200 |
|---|---|---|---|---|---|---|---|---|
| 2.28094 | 3.11749 | 4.69951 | 8.08982 | 17.1978 | 24.7959 | 37.9716 | 49.7614 | 66.0482 |

Table 22: Second part of timings for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 | 6120 | 6360 |
|---|---|---|---|---|---|---|---|---|
| 78.8248 | 95.48 | 111.111 | 132.495 | 156.039 | 182.562 | 194.174 | 228.423 | 253.517 |

Table 23: Third part of timings for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

| 6600 | 6840 | 7080 | 7320 |
|---|---|---|---|
| 289.097 | 318.358 | 357.77 | 422.593 |

Table 24: Last part of timings for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the runtimes in seconds.

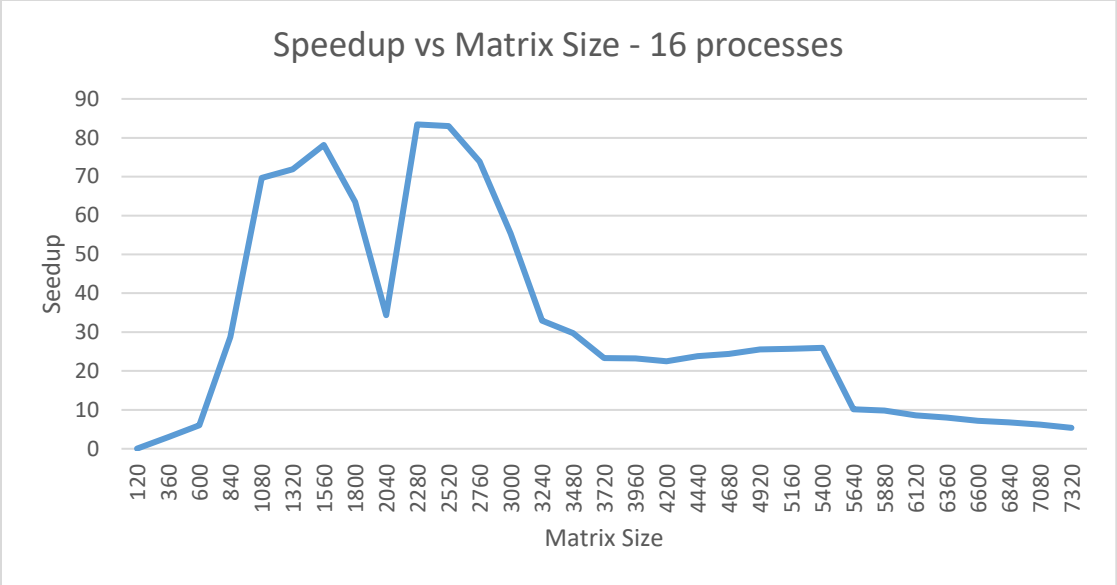


Fig. 10 Graph of speedup with different sizes of the matrices for a 16 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 | 2280 | 2520 | 2760 | 3000 | 3240 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.04 | 3 | 6.08 | 28.8 | 69.71 | 71.91 | 78.14 | 63.49 | 34.4 | 83.44 | 83.03 | 73.84 | 55.32 | 32.99 |

Table 25: First part of speedup for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

| 3480 | 3720 | 3960 | 4200 | 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 | 6120 | 6360 | 6600 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 29.72 | 23.34 | 23.27 | 22.55 | 23.85 | 24.4 | 25.56 | 25.74 | 25.96 | 10.13 | 9.83 | 8.62 | 8 | 7.23 |

Table 26: Second part of speedup for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

| 6840 | 7080 | 7320 |
|---|---|---|
| 6.75 | 6.17 | 5.37 |

Table 27: Last part of speedup for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the speedup

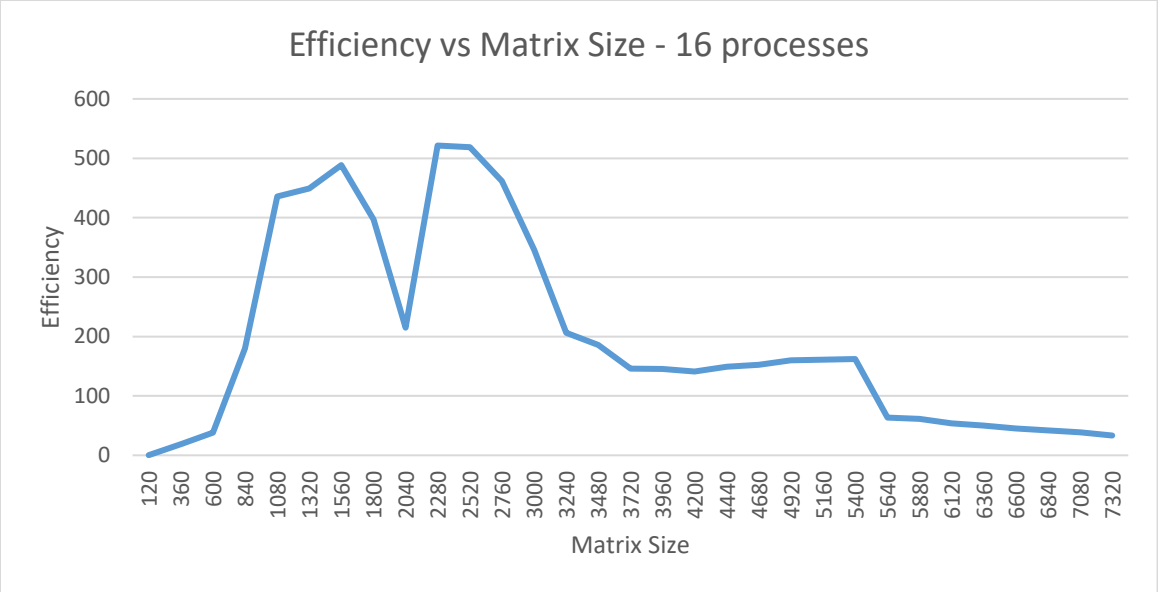


Fig. 11 Graph of efficiency with different sizes of the matrices for a 16 process mesh.

| 120 | 360 | 600 | 840 | 1080 | 1320 | 1560 | 1800 | 2040 | 2280 | 2520 | 2760 | 3000 | 3240 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.24 | 18.73 | 38.03 | 179.98 | 435.71 | 449.41 | 488.37 | 396.82 | 214.97 | 521.49 | 518.95 | 461.51 | 345.76 | 206.17 |

Table 28: First part of efficiency for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

| 3480 | 3720 | 3960 | 4200 | 4440 | 4680 | 4920 | 5160 | 5400 | 5640 | 5880 | 6120 | 6360 | 6600 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 185.74 | 145.9 | 145.46 | 140.92 | 149.06 | 152.52 | 159.75 | 160.86 | 162.22 | 63.3 | 61.45 | 53.87 | 50.02 | 45.16 |

Table 29: Second part of efficiency for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

| 6840 | 7080 | 7320 |
|---|---|---|
| 42.19 | 38.59 | 33.56 |

Table 30: Last part of efficiency for matrix multiplication calculations with 16 processes, with the top row being the sizes of the matrices and the bottom row indicating the efficiency

# Analysis

**Sequential**

As for the sequential algorithm, the main goal was to find out how the runtimes would look with different square matrix sizes, up to five minutes. With the $O(n^3)$ algorithm being used for matrix multiplication, it got to five minute runtimes at around the 2600x2600 matrix sizes and up to 24 minute runtimes around 4200x4200. The estimated times beyond that were calculated by adding a minute on each time, since the difference between the last non-estimated time and its previous time was a minute.

It was around 3240x3240 that the sequential times had a sharp jump, where previously it was a relatively smooth curve. This is likely when the matrix size exceeded the cache size for the process.

**Parallel** – 4 processes

The parallel implementation with 4 processes was much faster than the sequential, with superlinear speedup from 360x360 to 4400x4400. Interestingly, from 360x360 to 1320x1320 the speedup sharply increased, with a very small dip between 840x840 and 1080x1080. Afterwards, the speedup sharply decreased to just above 4. I assume the sharp increase was due to the improved runtimes of the parallel algorithm, and without the cache situation causing superlinear speedup it would still be a sharp increase non-superlinear. The dip in speedup and efficiency is likely due to communication times being less worth it for that specific range of matrix sizes. I suspect that beyond 4400x4400, the matrix size of the chunks exceeded the cache size for the individual processes. Even so, the speedup was still just below 4 for the sizes beyond that.

**Parallel** – 9 processes

The parallel implementation with 9 processes was even faster than the 4 processes, with superlinear speedup from 840x840 to 5400x5400. Similar to the 4 process implementation, from 600x600 to 2040x2040 the speedup sharply increased, with a larger dip than the 4 proccess implementation between 1560x1560 and 1800x1800. Afterwards, the speedup sharply decreased to just above 9. I again assume the sharp increase was due to the improved runtimes of the parallel algorithm, and without the cache situation causing superlinear speedup it would still be a sharp increase non-superlinear. The dip in speedup and efficiency is likely due to communication times being less worth it for that specific range of matrix sizes. This explains why the dip is greater with 9 processes than with 4 processes. I suspect that beyond 5400x5400, the matrix size of the chunks exceeded the cache size for the individual processes, causing speedup to no longer be superlinear.

**Parallel** – 16 processes

The parallel implementation with 16 processes was even faster than the 9 processes, with superlinear speedup from 840x840 to 5400x5400, just like the 9 process implementation. Similar to the 4 and 9 process implementations, from 600x600 to 2280x2280 the speedup sharply increased, with a very large dip between 1560x1560 and 2040x2040. Afterwards, the speedup sharply decreased to just above 20. I again assume the sharp increase was due to the improved runtimes of the parallel algorithm, and without the cache situation causing superlinear speedup it would still be a sharp increase non-superlinear. The dip in speedup and efficiency is very likely due to communication times being less

worth it for that specific range of matrix sizes. This explains why the dip is greatest with 16 processes than with 4 or 9 processes. I suspect that beyond 5400x5400, the matrix size of the chunks exceeded the cache size for the individual processes, causing speedup to no longer be superlinear.