PA4: Matrix Multiplicatio

Evan Grill

CS 415

April 13, 2017

Introduction

For this project, we were tasked to implement matrix multiplication both sequentially, and in parallel. For the parallel version, we were to use Cannon's algorithm.

Sequential

Problem

Implement matrix multiplication sequentially.

Procedure

Each matrix is represented by a 2D C-style array. To multiply them, two loops are used to iterate through each entry in the result. For each entry, a third loop is run, and the multiplications of the corresponding rows/columns are added and stored.

Data

Data Size	Execution Time
120	0.008901
240	0.02199
360	0.089177
480	0.245352
600	0.450391
720	0.882469
840	3.934625
960	9.890512
1080	16.545319
1200	23.962681
1320	33.170678
1440	41.535923
1560	55.148374
1680	69.693656
1800	88.416437
1920	112.95418
2040	129.002905
2160	156.119676
2280	183.674783
2400	211.609911

2520	243.277265
2640	287.558517
2760	330.812922
2880	415.161814
3000	425.957607

Table 1: Execution time (in seconds) of sequential matrix multiplication. (Matrix is of size x size total entries)

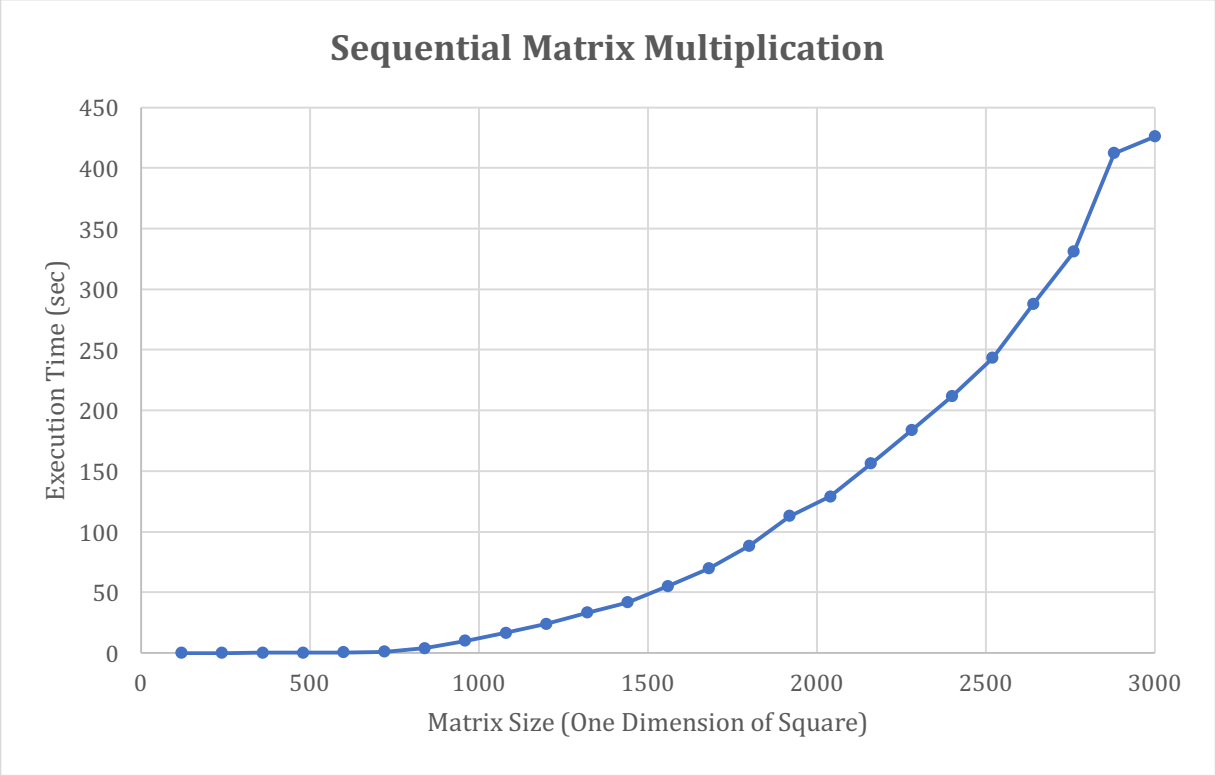

Figure 1: Chart showing execution times from Table 1 showing polynomial growth.

Results

The project presently only has the sequential method implemented, so there is not any data to compare it to, but it displays an expected polynomial growth rate. The algorithm is $O(n^3)$, so the growth rate reflects that. As always, we're tasked with determining the point at which the algorithm reaches five minute (300 seconds) run time. In this case, that point would be about 2700x2700.

The next portion of the project and report will add parallel implementation with comparisons to sequential, speed up factors, and efficiencies, for multiple amounts of parallel processors.

Note: The timings for this version of the report differ from Part 1. They were adjusted to fit sizes appropriate for parallel computation (multiples of 60).

Parallel

Problem

Implement parallel multiplication using Cannon's Algorithm. Compare execution times for different amounts of processors and different matrix sizes.

Procedure

For this portion of the project, a perfect square number of parallel processors are laid out in a grid pattern logically. The two matrices to be multiplied are divided up and distributed to the matching processor in the grid as seen in Figure 2.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix} \rightarrow \begin{matrix} \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 4 & 5 \end{bmatrix} \\ task\ 0 & task\ 1 \\ \begin{bmatrix} 3 & 4 \\ 4 & 5 \end{bmatrix} & \begin{bmatrix} 5 & 6 \\ 6 & 7 \end{bmatrix} \\ task\ 2 & task\ 3 \end{matrix}$$

Figure 2: Demonstration of a 4x4 matrix split among 4 parallel processors.

After distribution, the matrices are initialized. Initialization consists of each row of the processor grid shifting their left submatrix left and each column of the processor grid shifting their right submatrix up by which row or column they are in in the processor grid (Row 0 shifts 0, row 1 shifts 1, etc.). See figure 3 for an example.

$$\begin{matrix} \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 4 & 5 \end{bmatrix} \\ task\ 0 & task\ 1 \\ \begin{bmatrix} 5 & 6 \\ 6 & 7 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 4 & 5 \end{bmatrix} \\ task\ 2 & task\ 3 \end{matrix}$$

Figure 3: Left side matrix initialized. (Bottom row is shifted 1 left)

Following the initialization, matrix multiplication is performed sequentially on each submatrix. The result is stored in an accumulator matrix on each processor. Then every row of the processor grid shifts its left submatrix one spot left and every column of the processor grid shifts its right submatrix one spot up (Note: no longer based on row).

The previous paragraph is performed the same number of times as is the size of one dimension of the processor grid (2 times in our examples from Figure 2 and Figure 3). Once done, the accumulator matrices are all transmitted back to the main task for creation of the solution to be output as the user desires.

Data

Timing, speedup, and efficiency data will come in the next part of this report.

Results

After the code review performed by my classmates, several improvements were made to the code. Firstly, memory management was improved by reducing the number of duplicate matrices generated and properly deallocating matrices after their usefulness ended. Next, MPI_Cart_shift( ) was implemented rather than the more cumbersome MPI_Cart_rank( ). Lastly input from and output to text files as structured in class was implemented.

Analysis of data will come in a future version of this report.