PA4: Matrix Multiplication

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)

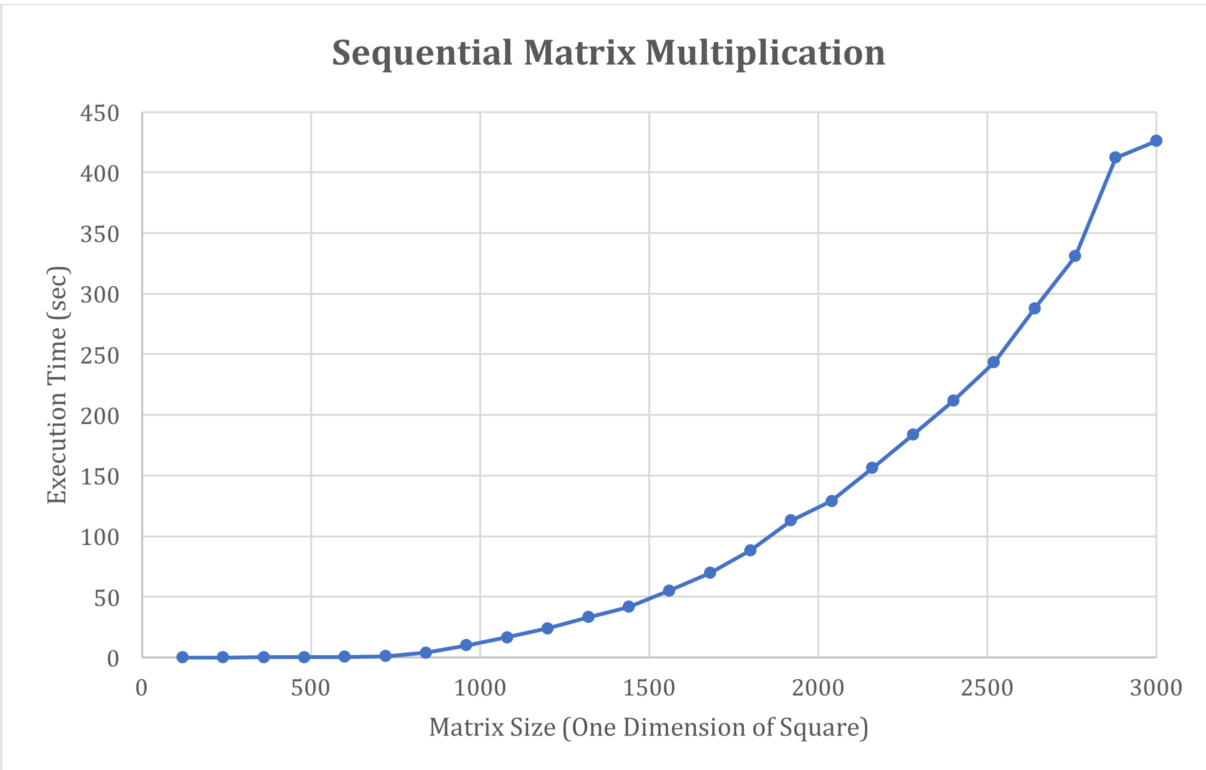

Figure 1: Chart showing execution times from Table 1 showing polynomial growth.

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

Size	1 Proc	4 Proc	9 Proc	16 Proc	25 Proc
120	0.008901	0.002645	0.003144	0.011387	0.013461
240	0.021990	0.012240	0.008178	0.019940	0.016999
360	0.089177	0.023869	0.022860	0.014307	0.017781
480	0.245352	0.053190	0.040415	0.029167	1.060540
600	0.450391	0.100286	0.080701	0.046170	0.725646
720	0.882469	0.189959	0.116562	0.101201	0.901156
840	3.934625	0.499173	0.178848	0.144073	2.281232
960	9.890512	0.507609	0.353053	0.142397	1.037606
1080	16.545319	0.701604	0.367018	0.223966	0.594409
1200	23.962681	0.914804	0.480283	0.241513	1.305976
1320	33.170678	1.722446	0.612235	0.354312	1.492221
1440	41.535923	1.795622	0.898631	0.478034	2.352583
1560	55.148374	5.289325	1.114496	0.897539	1.827983
1680	69.693656	8.053441	1.292996	1.147857	1.912707
1800	88.416437	13.713929	1.604953	1.014078	1.640548
1920	112.954180	20.384662	3.322088	1.228423	2.253064
2040	129.002905	29.180260	2.578973	1.543434	2.129833
2160	156.119676	34.645002	2.976704	1.593851	2.047257
2280	183.674783	45.622651	5.566730	2.050441	2.820567
2400	211.609911	49.539171	11.283316	2.120212	2.604509
2520	243.277265	59.516475	13.613704	3.969672	3.900804
2640	287.558517	71.589665	19.951446	4.872169	4.071604
2760	330.812922	75.628049	26.235731	5.319170	3.943985
2880	484.496640	83.834260	31.605393	4.752741	5.483106
3000	548.100000	106.171857	38.168562	11.775302	3.589598
3120	616.973760	110.622686	45.428545	13.330742	4.940483
3240	691.325280	134.096443	52.717769	18.323711	5.015167
3360	771.361920	143.019887	58.259395	18.862127	7.783586
3480	857.291040	167.351885	68.371664	33.162067	6.610134
3600	949.320000	181.504484	74.583226	30.190383	7.562287
3720	1047.656160	206.459282	84.208644	42.439920	9.177727
3840	1152.506880	227.373706	94.780526	41.598264	18.635478
3960	1264.079520	239.125826	103.546437	50.135215	15.894380
4080	1382.581440	278.603646	113.902207	61.250071	19.298353
4200	1508.220000	286.384362	124.517169	63.370937	23.843994
4320	1641.202560	313.473592	129.496633	69.338754	29.638568
4440	1781.736480	374.178336	150.852643	76.486001	35.041644
4560	1930.029120	405.632064	161.214840	88.438093	39.838167
4680	2086.287840	438.748128	174.338985	98.854445	49.311119
4800	2250.720000	473.568000	199.812244	98.465943	54.871047

4920	2423.532960	510.133152	206.258141	120.437495	60.765030
5040	2604.934080	548.485056	215.679246	121.031613	67.757148
5160	2795.130720	588.665184	235.952994	140.002349	76.152264
5280	2994.330240	630.715008	254.680743	135.726007	85.427127
5400	3202.740000	674.676000	289.141775	145.053131	89.086589
5520	3420.567360	720.589632	297.185928	169.145237	95.680450
5640	3648.019680	768.497376	314.315196	166.068549	102.563839
5760	3885.304320	818.440704	362.207232	170.221781	116.538096
5880	4132.628640	870.461088	386.391264	214.280858	119.018069
6000	4390.200000	924.600000	411.600000	215.676859	126.393915
6120	4658.225760	980.898912	437.854176	238.261362	135.893569
6240	4936.913280	1039.399296	465.174528	243.719077	138.711536
6360	5226.469920	1100.142624	493.581792	274.085486	158.970021
6480	5527.103040	1163.170368	523.096704	289.625123	164.760403
6600	5839.020000	1228.524000	553.740000	298.501699	182.714539
6720	6162.428160	1296.244992	585.532416	290.323604	180.600085
6840	6497.534880	1366.374816	618.494688	306.544254	193.274230
6960	6844.547520	1438.954944	652.647552	312.041856	206.554582
7080	7203.673440	1514.026848	688.011744	329.520192	217.698148
7200	7575.120000	1591.632000	724.608000	347.616000	216.681803
7320	7959.094560	1671.811872	762.457056	366.339648	241.684475
7440	8355.804480	1754.607936	801.579648	385.701504	253.851793
7560	8765.457120	1840.061664	841.996512	405.711936	271.219803
7680	9188.259840	1928.214528	883.728384	426.381312	285.976603
7800	9624.420000	2019.108000	926.796000	447.720000	292.291256
7920	10074.144960	2112.783552	971.220096	469.738368	315.986745

Table 2: Execution times (in seconds) for sequential and parallel matrix multiplication. Values in red are extrapolations.

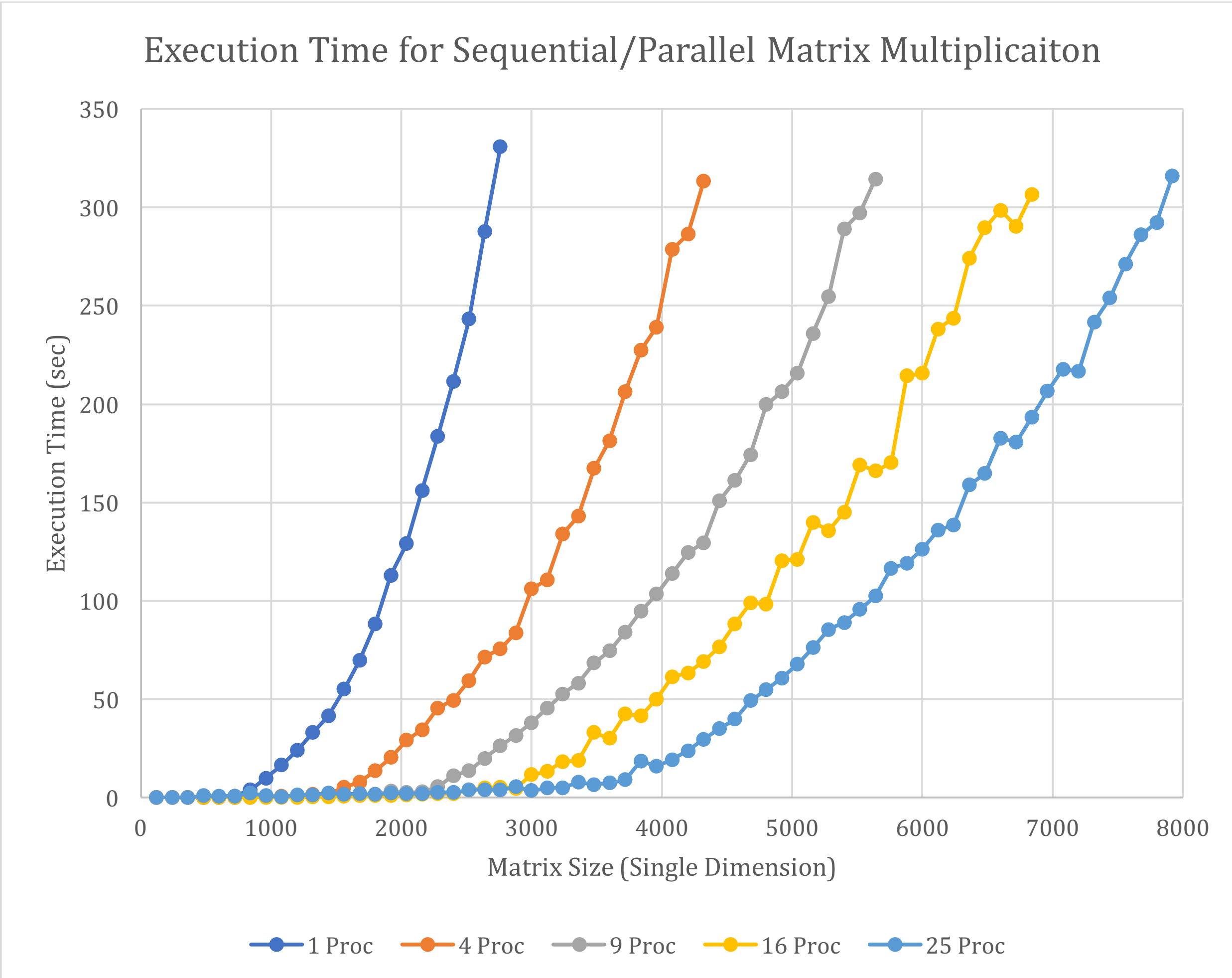

Figure 4: Chart showing execution times from Table 2. Extrapolated times from Table 2 are not included in this chart.

Size	1 Proc	4 Proc	9 Proc	16 Proc	25 Proc
120	1	3.365217391	2.83110687	0.781680864	0.661243593
240	1	1.796568627	2.688921497	1.102808425	1.293605506
360	1	3.736101219	3.901006124	6.233102677	5.015297227
480	1	4.612746757	6.070815291	8.411972435	0.231346295
600	1	4.491065553	5.580984127	9.755057397	0.620675922
720	1	4.645576151	7.5708121	8.719963241	0.979263302
840	1	7.882287303	21.99982667	27.3099401	1.724780733
960	1	19.48450875	28.01424149	69.45730598	9.532049738
1080	1	23.58213323	45.08040205	73.8742443	27.8349066
1200	1	26.19433343	49.8928361	99.21901098	18.34848496
1320	1	19.25789139	54.17964997	93.61996771	22.22906527
1440	1	23.13177439	46.22133334	86.88905601	17.65545488
1560	1	10.42635384	49.48279222	61.44398628	30.16897531
1680	1	8.65389788	53.90090611	60.71632268	36.43718353

1800	1	6.447199559	55.08973596	87.18899039	53.89445295
1920	1	5.541135781	34.00095964	91.95055775	50.13358697
2040	1	4.420896353	50.02103744	83.5817437	60.56949301
2160	1	4.50626835	52.44716169	97.95123635	76.2579764
2280	1	4.025955945	32.99509461	89.57818489	65.11980853
2400	1	4.271567463	18.75423067	99.80601515	81.24752535
2520	1	4.087561721	17.87002751	61.28397132	62.36592892
2640	1	4.016760199	14.41291609	59.0206368	70.6253646
2760	1	4.374209389	12.60925118	62.19258305	83.87783473
2880	1	5.779220095	15.32955594	101.9404676	88.36171323
3000	1	5.162384981	14.35998558	46.54657689	152.6911927
3120	1	5.577280595	13.5811913	46.28202691	124.8812636
3240	1	5.155433392	13.11370517	37.72845359	137.8469112
3360	1	5.393389242	13.24012925	40.89474745	99.10109813
3480	1	5.122685293	12.53868913	25.85155624	129.6934434
3600	1	5.230284008	12.72833117	31.44445037	125.533453
3720	1	5.074396025	12.44119499	24.68562994	114.1520291
3840	1	5.068778181	12.15974345	27.70564849	61.84477157
3960	1	5.286252602	12.20785144	25.21340579	79.5299672
4080	1	4.962538932	12.13832002	22.57273204	71.64245778
4200	1	5.266418842	12.1125465	23.79986902	63.25366463
4320	1	5.235536906	12.67370836	23.66934024	55.37388176
4440	1	4.761730727	11.81110549	23.29493576	50.84625824
4560	1	4.758078296	11.97178324	21.82350449	48.44673501
4680	1	4.755092288	11.96684631	21.1046437	42.30866957
4800	1	4.752685992	11.26417458	22.85785249	41.01835345
4920	1	4.750785066	11.74999905	20.12274467	39.88367915
5040	1	4.749325531	12.07781522	21.52275769	38.44515534
5160	1	4.748252141	11.84613373	19.96488445	36.70449929
5280	1	4.747517028	11.75719139	22.06158058	35.05128108
5400	1	4.747078598	11.07671142	22.07977158	35.95086574
5520	1	4.746900605	11.50985642	20.22266438	35.74990878
5640	1	4.746951381	11.60624662	21.96695101	35.56828328
5760	1	4.747203189	10.72674419	22.82495399	33.33934956
5880	1	4.747631683	10.69545051	19.28603739	34.72269946
6000	1	4.748215445	10.66618076	20.35545223	34.73426707
6120	1	4.748935597	10.63876061	19.55090713	34.27848569
6240	1	4.74977547	10.61303443	20.25657302	35.5912235
6360	1	4.750720321	10.58886289	19.06875842	32.87707888
6480	1	4.751757087	10.56612094	19.08364503	33.54630688
6600	1	4.752874181	10.54469607	19.56109469	31.95706281
6720	1	4.754061306	10.52448676	21.22606662	34.12195603
6840	1	4.755309307	10.50540127	21.19607461	33.61821636

6960	1	4.75661003	10.48735646	21.93470968	33.13674988
7080	1	4.757956208	10.47027686	21.8610987	33.09019165
7200	1	4.759341355	10.4540938	21.79163215	34.95965003
7320	1	4.760759684	10.43874471	21.72599827	32.93175766
7440	1	4.762206023	10.42417245	21.66391469	32.9160743
7560	1	4.763675746	10.41032474	21.60512507	32.31864717
7680	1	4.765164719	10.3971537	21.54939624	32.12941109
7800	1	4.766669242	10.38461538	21.49651568	32.92749886
7920	1	4.768186003	10.37266939	21.44628935	31.88154288

Table 3: Speedup factor for sequential and parallel matrix multiplication. Values in red are from extrapolated data.

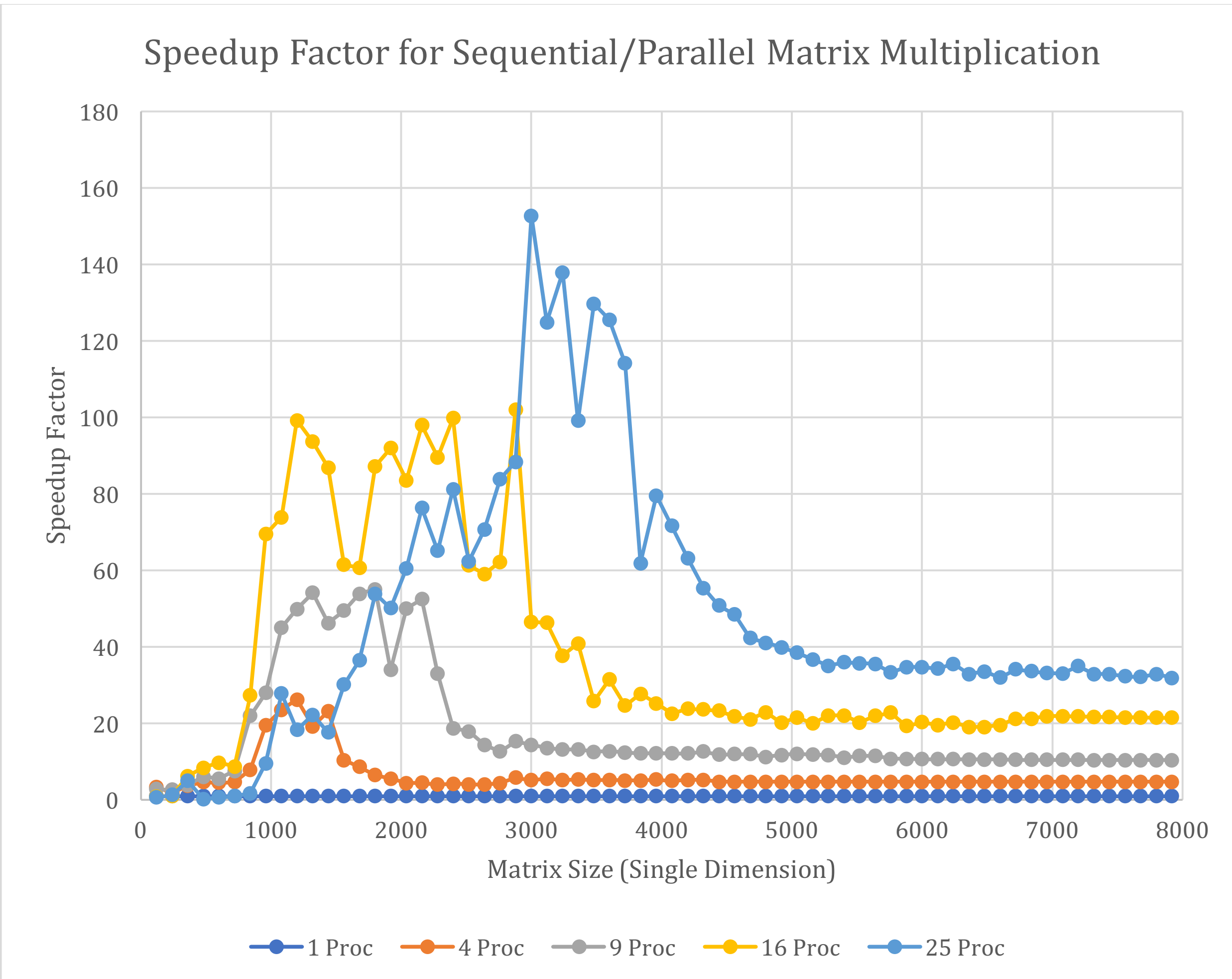

Figure 5: Chart showing speedup factors from Table 3. Extrapolated values are included in this chart to show trends toward linear speedup at larger matrix sizes.

Size	1 Proc	4 Proc	9 Proc	16 Proc	25 Proc
120	1	84.13%	31.46%	4.89%	2.64%
240	1	44.91%	29.88%	6.89%	5.17%

360	1	93.40%	43.34%	38.96%	20.06%
480	1	115.32%	67.45%	52.57%	0.93%
600	1	112.28%	62.01%	60.97%	2.48%
720	1	116.14%	84.12%	54.50%	3.92%
840	1	197.06%	244.44%	170.69%	6.90%
960	1	487.11%	311.27%	434.11%	38.13%
1080	1	589.55%	500.89%	461.71%	111.34%
1200	1	654.86%	554.36%	620.12%	73.39%
1320	1	481.45%	602.00%	585.12%	88.92%
1440	1	578.29%	513.57%	543.06%	70.62%
1560	1	260.66%	549.81%	384.02%	120.68%
1680	1	216.35%	598.90%	379.48%	145.75%
1800	1	161.18%	612.11%	544.93%	215.58%
1920	1	138.53%	377.79%	574.69%	200.53%
2040	1	110.52%	555.79%	522.39%	242.28%
2160	1	112.66%	582.75%	612.20%	305.03%
2280	1	100.65%	366.61%	559.86%	260.48%
2400	1	106.79%	208.38%	623.79%	324.99%
2520	1	102.19%	198.56%	383.02%	249.46%
2640	1	100.42%	160.14%	368.88%	282.50%
2760	1	109.36%	140.10%	388.70%	335.51%
2880	1	144.48%	170.33%	637.13%	353.45%
3000	1	129.06%	159.56%	290.92%	610.76%
3120	1	139.43%	150.90%	289.26%	499.53%
3240	1	128.89%	145.71%	235.80%	551.39%
3360	1	134.83%	147.11%	255.59%	396.40%
3480	1	128.07%	139.32%	161.57%	518.77%
3600	1	130.76%	141.43%	196.53%	502.13%
3720	1	126.86%	138.24%	154.29%	456.61%
3840	1	126.72%	135.11%	173.16%	247.38%
3960	1	132.16%	135.64%	157.58%	318.12%
4080	1	124.06%	134.87%	141.08%	286.57%
4200	1	131.66%	134.58%	148.75%	253.01%
4320	1	130.89%	140.82%	147.93%	221.50%
4440	1	119.04%	131.23%	145.59%	203.39%
4560	1	118.95%	133.02%	136.40%	193.79%
4680	1	118.88%	132.96%	131.90%	169.23%
4800	1	118.82%	125.16%	142.86%	164.07%
4920	1	118.77%	130.56%	125.77%	159.53%
5040	1	118.73%	134.20%	134.52%	153.78%
5160	1	118.71%	131.62%	124.78%	146.82%
5280	1	118.69%	130.64%	137.88%	140.21%
5400	1	118.68%	123.07%	138.00%	143.80%

5520	1	118.67%	127.89%	126.39%	143.00%
5640	1	118.67%	128.96%	137.29%	142.27%
5760	1	118.68%	119.19%	142.66%	133.36%
5880	1	118.69%	118.84%	120.54%	138.89%
6000	1	118.71%	118.51%	127.22%	138.94%
6120	1	118.72%	118.21%	122.19%	137.11%
6240	1	118.74%	117.92%	126.60%	142.36%
6360	1	118.77%	117.65%	119.18%	131.51%
6480	1	118.79%	117.40%	119.27%	134.19%
6600	1	118.82%	117.16%	122.26%	127.83%
6720	1	118.85%	116.94%	132.66%	136.49%
6840	1	118.88%	116.73%	132.48%	134.47%
6960	1	118.92%	116.53%	137.09%	132.55%
7080	1	118.95%	116.34%	136.63%	132.36%
7200	1	118.98%	116.16%	136.20%	139.84%
7320	1	119.02%	115.99%	135.79%	131.73%
7440	1	119.06%	115.82%	135.40%	131.66%
7560	1	119.09%	115.67%	135.03%	129.27%
7680	1	119.13%	115.52%	134.68%	128.52%
7800	1	119.17%	115.38%	134.35%	131.71%
7920	1	119.20%	115.25%	134.04%	127.53%

Table 4: Efficiency for sequential and parallel matrix multiplication. Values in red are extrapolations.

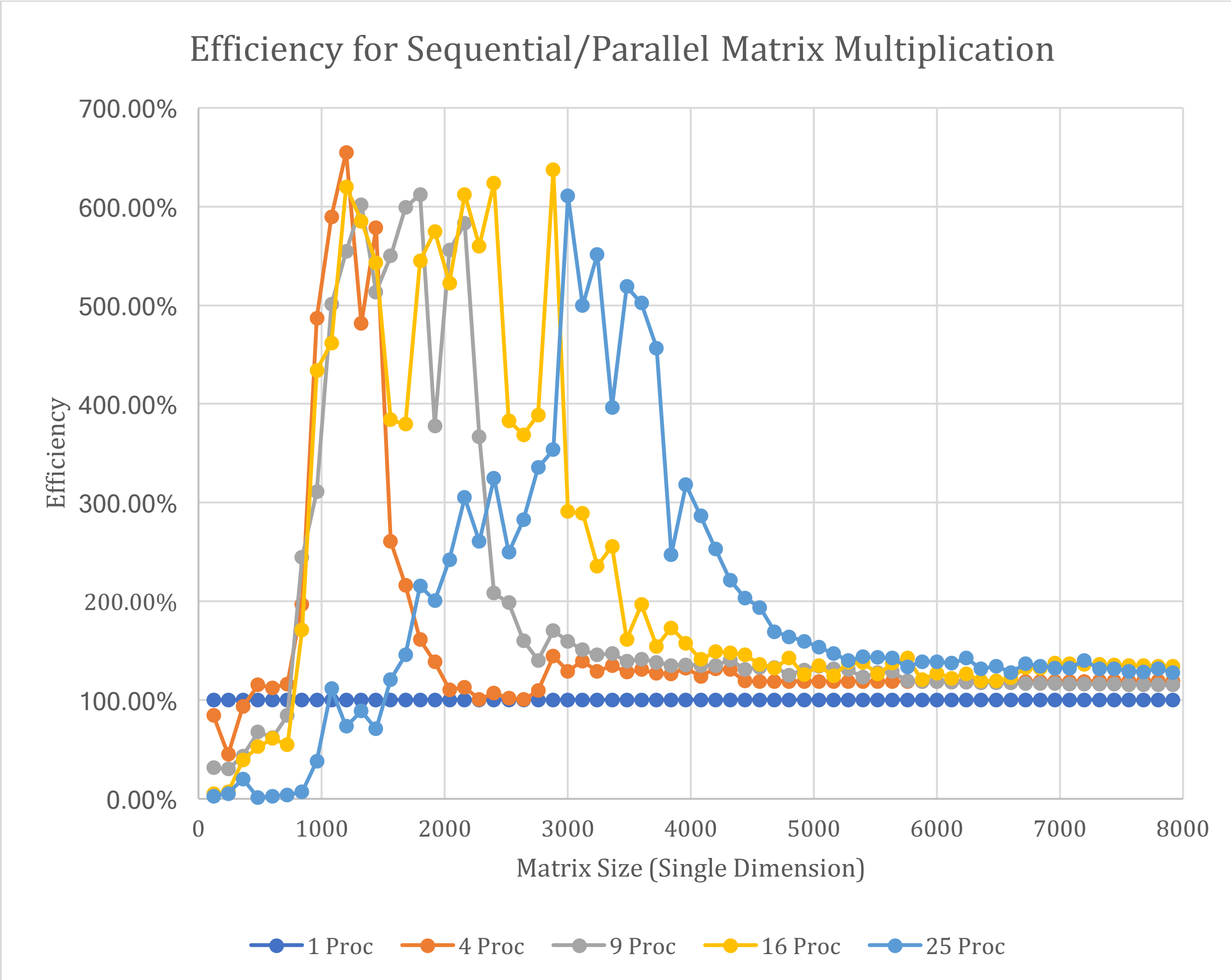

Figure 6: Chart showing efficiency values from Table 4. Extrapolated data is included to show trend toward 100% Efficiency

Results

As can be seen in the above data, increasing the number of processors did reduce execution times for similarly sized matrices. In fact, for lower size matrices, you can see superlinear speedup when implementing Cannon's algorithm on parallel processors. As the size of the matrices increase the speed up drops to linear improvements.

As we discussed in class, smaller matrices can fit entirely in cache. This improves the individual multiplications execution time greatly. This is why we see superlinear speedup in smaller matrices. The sequential multiplication at these sizes is too large to fit in cache, so memory has to be shifted back and forth causing slowdown. However, on the parallel processors, we've divided these larger matrices into smaller matrices that still fit into cache.

It's important to note that while the execution time is decreased for larger numbers of parallel processors, it does not reduce the asymptotic behavior of the algorithm. Because we are still performing sequential matrix multiplication on the pieces, we still have an algorithm that runs at $O(n^3)$.

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.