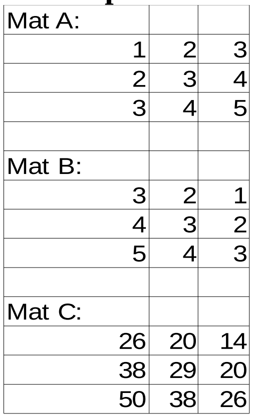
Andrew Frost
CS415
PA4 Report
05/03/2017

Overview:

In this programming assignment matrices were multiplied using sequential and parallel algorithms. The sequential program made us of a simplistic matrix multiplication algorithm that ran in O(n^3). Cannon's algorithm was used in the parallel program. In this report the parallel run times and sequential run times are compared. The timing data for both the parallel and sequential program are made up of discrete samples. The complete data for this report can be found in data.xlsx. Table 1 shows a sample result of matrix multiplication.

Matrix Multiplication Results



Mat A:		
1	2	3
2	3	4
3	4	5
Mat B:		
3	2	1
4	3	2
5	4	3
Mat C:		
26	20	14
38	29	20
50	38	26

Table 1: an example of multiplying two 3x3 matrices using the sequential algorithm.

Table 1 shows a sample result from multiplying two matrices together using the sequential algorithm. The sequential algorithm will be discussed in depth below.

Sequential:

The sequential algorithm run times are based off of single sample run times. On select tests, multiple time trials were ran and little deviation was found. Based on the low variance found in select sample run times, the data used in this report for sequential run times was not averaged or processed. The sequential algorithm used had a time complexity of O(n^3) and the run time results reflect this curve. Figure 1 graphs the run times of the sequential algorithm.

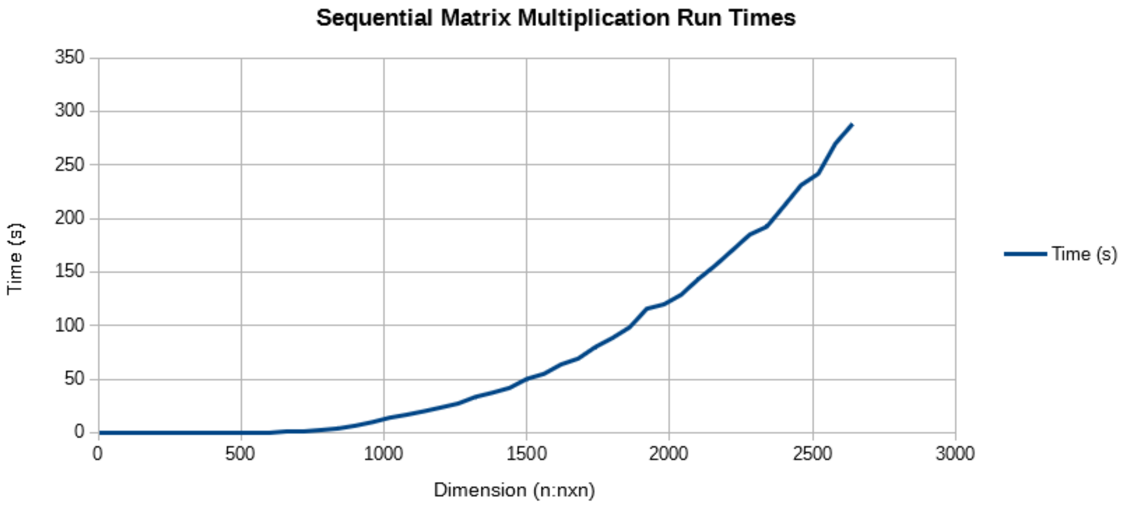
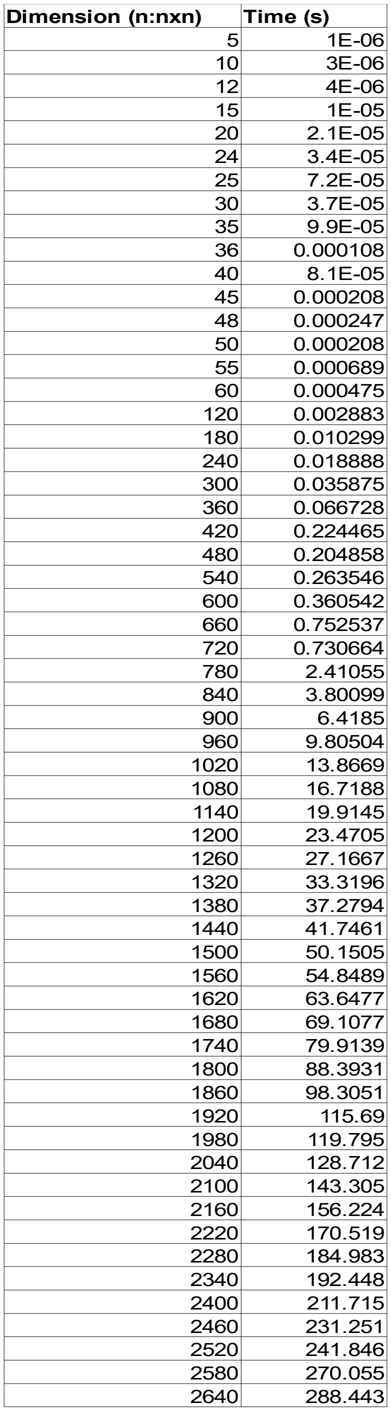


Figure 1: a graph of sequential run times in seconds. As the size of the matrix increases, the run time required increases dramatically.

As seen in the graph in Figure 1, the run time of the sequential algorithm increases dramatically as the matrix dimension size increases. The maximum tested dimension of 2640x2640 neared five minutes in run time. This not only shows the undesirability of an O(n^3), but highlights the need for an efficient parallel algorithm. The run times of the sequential algorithm are included in Table 2.

Sequential Matrix Multiplication Run Times



Dimension (n:nxn)	Time (s)
5	1E-06
10	3E-06
12	4E-06
15	1E-05
20	2.1E-05
24	3.4E-05
25	7.2E-05
30	3.7E-05
35	9.9E-05
36	0.000108
40	8.1E-05
45	0.000208
48	0.000247
50	0.000208
55	0.000689
60	0.000475
120	0.002883
180	0.010299
240	0.018888
300	0.035875
360	0.066728
420	0.224465
480	0.204858
540	0.263546
600	0.360542
660	0.752537
720	0.730664
780	2.41055
840	3.80099
900	6.4185
960	9.80504
1020	13.8669
1080	16.7188
1140	19.9145
1200	23.4705
1260	27.1667
1320	33.3196
1380	37.2794
1440	41.7461
1500	50.1505
1560	54.8489
1620	63.6477
1680	69.1077
1740	79.9139
1800	88.3931
1860	98.3051
1920	115.69
1980	119.795
2040	128.712
2100	143.305
2160	156.224
2220	170.519
2280	184.983
2340	192.448
2400	211.715
2460	231.251
2520	241.846
2580	270.055
2640	288.443

Table 2: the run time in seconds of the sequential matrix multiplication algorithm. The lack of fluctuation can be attributed to consistency provided by the algorithm running only on a single processor core of the same make and model.

Nothing unusual was encountered when testing the sequential algorithm. As expected, the sequential algorithm followed an O(n^3) curve as the size of the matrix increased. Notice that the run time nearly doubles from the previous tested matrix size when multiplying 780x780 matrices due to cache misses. The sequential algorithm will now be compared to the parallel algorithm.

Parallel:

Changes made due to Critiques:

The following changes were made to the parallel code based on feedback from the critiques:

- The code responsible for shifting columns up and shifting rows left were moved into two functions: ShiftUp( ) and ShiftLeft( ).
- The extra shifts up and right done after the last multiplication have been removed through the use of an if statement.
- Modular has been added to the code by creating functions for specific tasks in main.
    - The file I/O in main has been moved into functions.
    - The printing of the answer has been moved into functions.
    - The sending and receiving of sub-matrices in the initialization has been moved into functions.
- Some comments in main have been modified to be more descriptive and meaningful.

The changes made from the critiques have greatly increased the readability, modularity, and efficiency of the code for the parallel implementation.

Run Times:

The parallel implementation used Cannon's algorithm and saw impressively quick run times. The quick run times can be attributed largely to a higher instance of cache hits for the parallel implementation in comparison with the sequential algorithm. The parallel algorithm was tested at intervals of adding 60 per side between matrices with dimensions of 60x60 and 2640x2640 for square cores ranging from 4 to 25. Additional tests were run specifically for each square core size in that range. The parallel and sequential run times are directly compared in Fig. 2.

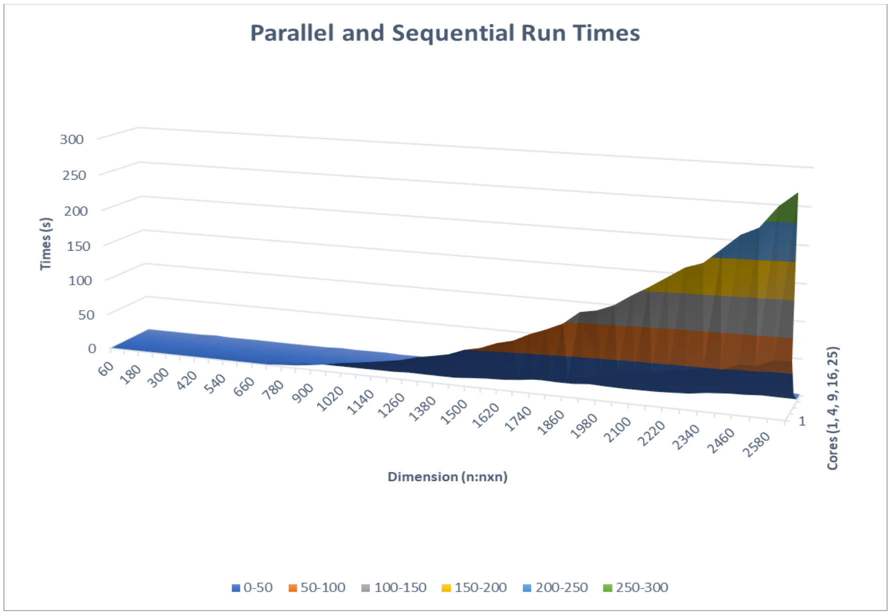
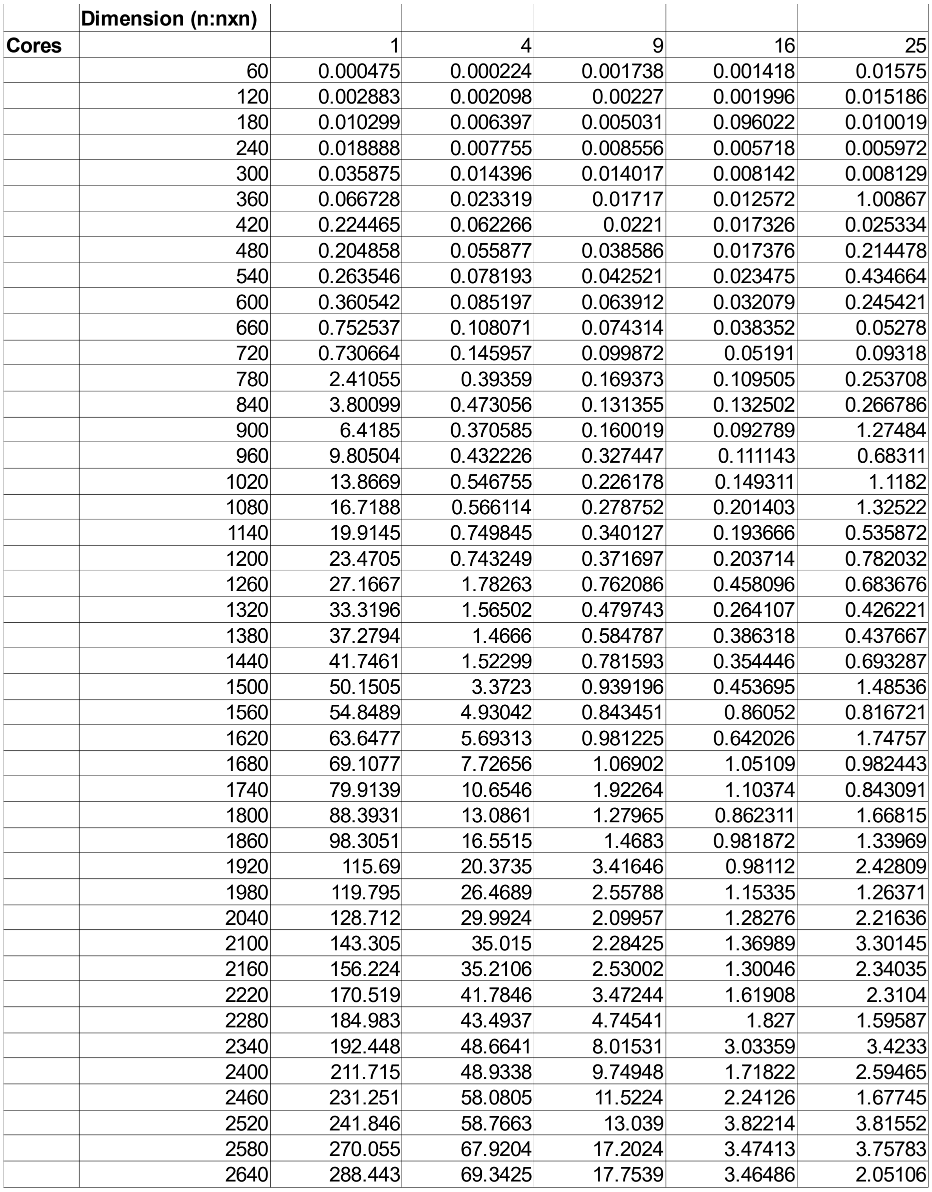


Figure 2: the parallel and sequential run times for matrix multiplication. The sequential time appears on this graph as the 1 core timings.

Figure 2 shows the parallel and sequential run times for matrix multiplication. The parallel time is small enough that it is difficult to see when placed next to the sequential run time in the plot. Table 3 shows a clearer picture of the parallel run times.

Run Time Per Number of Cores



	Dimension (n:nxn)					
Cores		1	4	9	16	25
	60	0.000475	0.000224	0.001738	0.001418	0.01575
	120	0.002883	0.002098	0.00227	0.001996	0.015186
	180	0.010299	0.006397	0.005031	0.096022	0.010019
	240	0.018888	0.007755	0.008556	0.005718	0.005972
	300	0.035875	0.014396	0.014017	0.008142	0.008129
	360	0.066728	0.023319	0.01717	0.012572	1.00867
	420	0.224465	0.062266	0.0221	0.017326	0.025334
	480	0.204858	0.055877	0.038586	0.017376	0.214478
	540	0.263546	0.078193	0.042521	0.023475	0.434664
	600	0.360542	0.085197	0.063912	0.032079	0.245421
	660	0.752537	0.108071	0.074314	0.038352	0.05278
	720	0.730664	0.145957	0.099872	0.05191	0.09318
	780	2.41055	0.39359	0.169373	0.109505	0.253708
	840	3.80099	0.473056	0.131355	0.132502	0.266786
	900	6.4185	0.370585	0.160019	0.092789	1.27484
	960	9.80504	0.432226	0.327447	0.111143	0.68311
	1020	13.8669	0.546755	0.226178	0.149311	1.1182
	1080	16.7188	0.566114	0.278752	0.201403	1.32522
	1140	19.9145	0.749845	0.340127	0.193666	0.535872
	1200	23.4705	0.743249	0.371697	0.203714	0.782032
	1260	27.1667	1.78263	0.762086	0.458096	0.683676
	1320	33.3196	1.56502	0.479743	0.264107	0.426221
	1380	37.2794	1.4666	0.584787	0.386318	0.437667
	1440	41.7461	1.52299	0.781593	0.354446	0.693287
	1500	50.1505	3.3723	0.939196	0.453695	1.48536
	1560	54.8489	4.93042	0.843451	0.86052	0.816721
	1620	63.6477	5.69313	0.981225	0.642026	1.74757
	1680	69.1077	7.72656	1.06902	1.05109	0.982443
	1740	79.9139	10.6546	1.92264	1.10374	0.843091
	1800	88.3931	13.0861	1.27965	0.862311	1.66815
	1860	98.3051	16.5515	1.4683	0.981872	1.33969
	1920	115.69	20.3735	3.41646	0.98112	2.42809
	1980	119.795	26.4689	2.55788	1.15335	1.26371
	2040	128.712	29.9924	2.09957	1.28276	2.21636
	2100	143.305	35.015	2.28425	1.36989	3.30145
	2160	156.224	35.2106	2.53002	1.30046	2.34035
	2220	170.519	41.7846	3.47244	1.61908	2.3104
	2280	184.983	43.4937	4.74541	1.827	1.59587
	2340	192.448	48.6641	8.01531	3.03359	3.4233
	2400	211.715	48.9338	9.74948	1.71822	2.59465
	2460	231.251	58.0805	11.5224	2.24126	1.67745
	2520	241.846	58.7663	13.039	3.82214	3.81552
	2580	270.055	67.9204	17.2024	3.47413	3.75783
	2640	288.443	69.3425	17.7539	3.46486	2.05106

Table 3: The run times for each dimension size for each node.

The use of the parallel algorithm was beneficial even when multiplying relatively small matrices of 60x60. Using 4 cores offered the best run times for multiplying matrices of 60x60. When multiplying matrices with dimensions of 240x240 to 1500x1500 16 cores offered the best performance. The important thing to note is that in a range of matrices with dimensions 60x60 to 2640x2640 the parallel implementation always ran quicker than the sequential algorithm. A more in depth analysis was done for each square core size tested on the parallel algorithm. Figure 3 shows the run times of the parallel algorithm when using 4 cores.

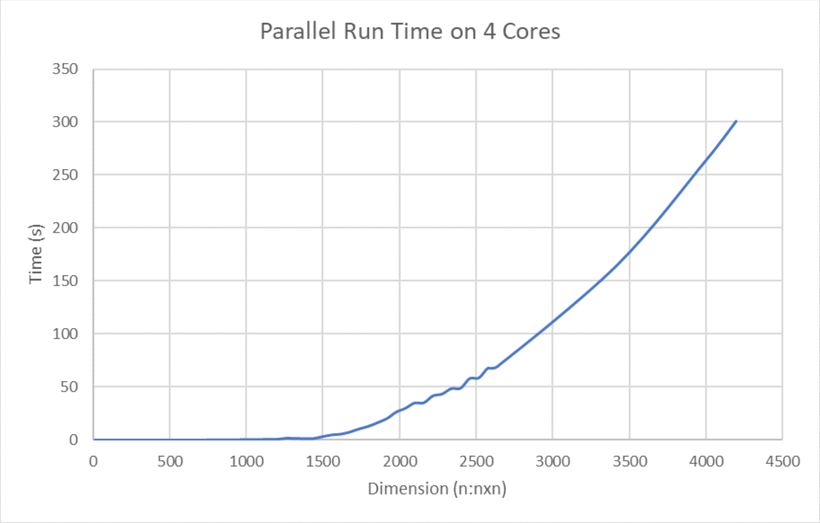


Figure 3: The parallel implementation's run times using 4 cores.

The parallel implementation was tested over a range of dimensions between 12x12 and 4200x4200. As shown in the graph, the impressive run times of the parallel algorithm fall victim to the O(n^3) computations required to multiply matrices as it surpasses 1500x1500 sized matrices. Table 4 displays the lower and upper bound run times for the parallel implementation on 4 cores.

Lower and Upper Bound Parallel Run Times on 4 Cores

Dimension (n:nxn)	Time (s)
12	6.1E-05
24	7.8E-05
36	0.00012
48	0.000145
60	0.000224
2520	58.7663
2580	67.9204
2640	69.3425
3420	164.726
4000	263.488
4200	300.832

Table 4: The run times of the upper and lower bounds of matrices multiplied using Cannon's Algorithm and 4 cores.

The parallel algorithm using 4 cores and the sequential algorithm have almost the same run time when multiplying matrices of 36x36. The parallel algorithm has a run time of 0.00012 and the sequential algorithm has a run of 0.000108; it isn't until matrices of 48x48 that the use of the parallel algorithm becomes worthwhile. The parallel algorithm using 4 cores reached 5 minutes when multiplying matrices of 4200x4200. The run times of the parallel algorithm using 9 cores is graphed in Fig. 4.

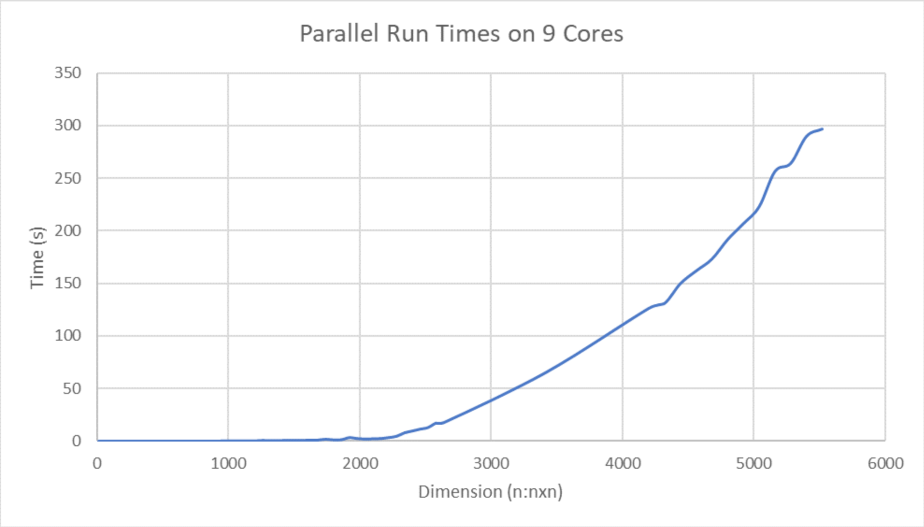


Figure 4:Parallel run times using 9 cores up to 5 minutes. Notice the sudden increase between 2000x2000 and 3000x3000 sized matrices.

As the extra cache gained from additional cores runs out, the run time of the algorithm begins to deteriorate to following O(n^3) computational time multiplied by a scalar to account for the division between cores. Table 5 shows the upper and lower bounds of run times for this implementation on 9 cores.

Lower and Upper Bound Parallel Run Times on 9 Cores

Dimension (n:nxn)	Time (s)
12	0.001608
24	0.001424
36	0.001946
48	0.001966
60	0.001738
2640	17.7539
3420	65.5759
4200	126.26
4320	130.947
4440	149.702
4560	161.752
4680	173.006
4800	191.671
4920	206.682
5040	222.867
5160	256.328
5280	263.961
5400	289.67
5520	296.503

Table 5: the upper and lower bound run time of the parallel implementation on 9 cores.

Between 2640x2640 and 3420x3420 there is a disproportionately high jump for this algorithm in the time required to multiply the matrices; the run time required increases by a factor of over 3. Between 3420x3420 and 4440x4440 the run time required only increases by a factor of only slightly over 2. This implies that the number of cache misses greatly increase between 2640x2940 and 3420x3420 for the parallel implementation on 9 cores. Figure 5 shows the run times of the parallel implementation on 16 cores.

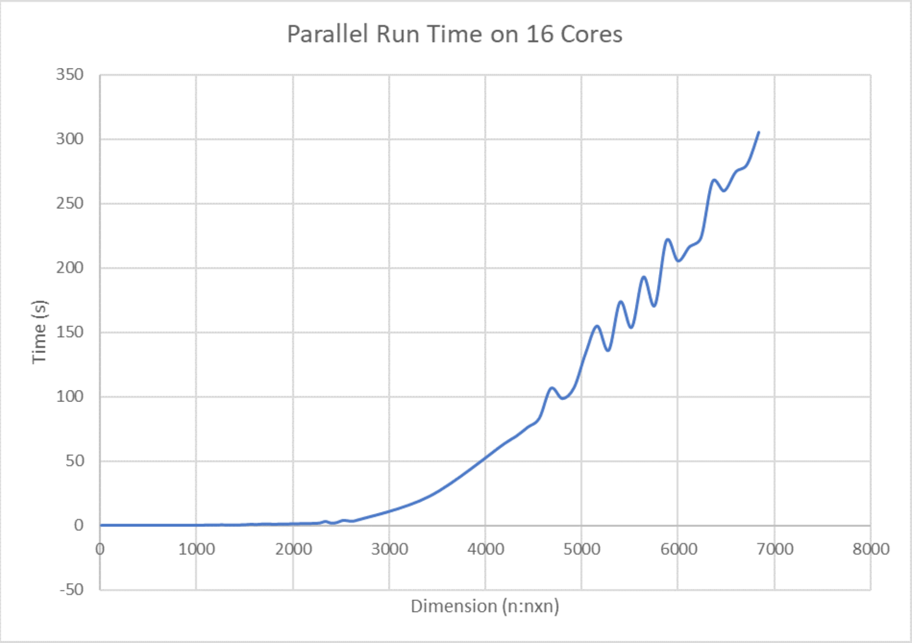
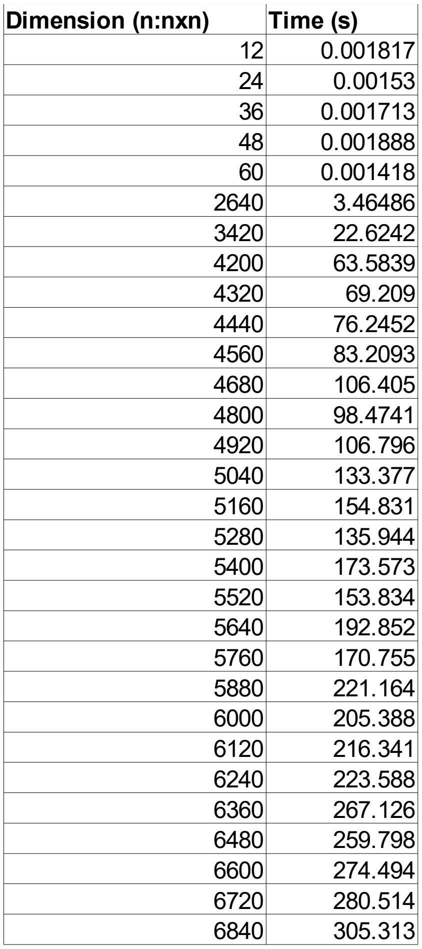


Figure 5: the parallel run time on 16 cores. Notice the spikes in run time between matrices of 4000x4000 and matrices of 7000x7000.

The dips in the run times in the upper range are a truly interesting phenomena. They are far too common to attribute to noise or cluster traffic. The run time of the algorithm began to degrade between 2000x2000 and 3000x3000 matrices. There is even a small amount of thrashing visible in this section. Table 6 contains the upper and lower bounds of run times for the parallel implementation with 16 cores and show cases the consistency of these peaks and valleys for matrices above 4000x4000.

Lower and Upper Bound Parallel Run Times on 16 Cores



Dimension (n:nxn)	Time (s)
12	0.001817
24	0.00153
36	0.001713
48	0.001888
60	0.001418
2640	3.46486
3420	22.6242
4200	63.5839
4320	69.209
4440	76.2452
4560	83.2093
4680	106.405
4800	98.4741
4920	106.796
5040	133.377
5160	154.831
5280	135.944
5400	173.573
5520	153.834
5640	192.852
5760	170.755
5880	221.164
6000	205.388
6120	216.341
6240	223.588
6360	267.126
6480	259.798
6600	274.494
6720	280.514
6840	305.313

Table 6: The upper and lower bounds of run times on the parallel implementation using 16 cores. On matrix sizes of 4000x4000 to 6600x6600 there is an almost sine wave pattern to the increasing run times.

The peaks and valleys in run times are caused by a certain ratio of cache hits and misses caused by specific matrix sizes. A similar pattern happens at varying sized matrices for each sized processor mesh. For 4 cores it occurs on matrices with dimensions between 780x780 and 1500x1500. For the 4 core implementation multiplying 780x780 matrices, which uses a 2x2 processor mesh, each processor stores 3 sub-matrices with dimensions of 390x390. When multiplying 1500x1500 matrices on 4 cores, each processor stores 3 sub-matrices with dimensions of 750x750. For 16 cores each processor would store 3 sub-matrices of 1290x1290 when multiplying matrices of 5160x5160. When multiplying matrices of 5280x5280 on 16 cores each processor stores 3 sub-matrices of 1320x1320, but has a quicker run time than multiplying matrices of 5160x5160. This implies that certain matrix sizes have a better cache hit ratio than others. This phenomena is less pronounced on odd numbers of processors that essentially have an entire extra node’s worth of cache. Figure 6 will display whether or not this trend is continued when using 25 cores.

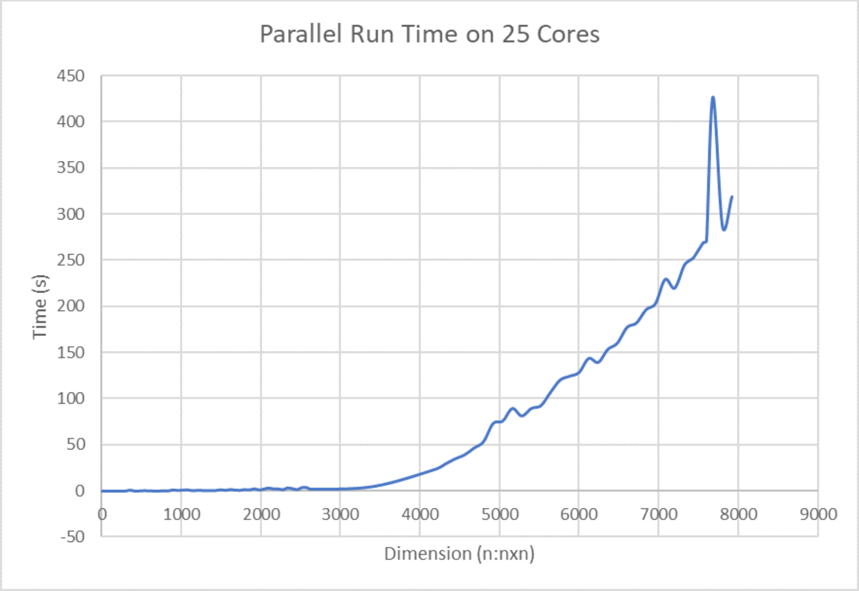
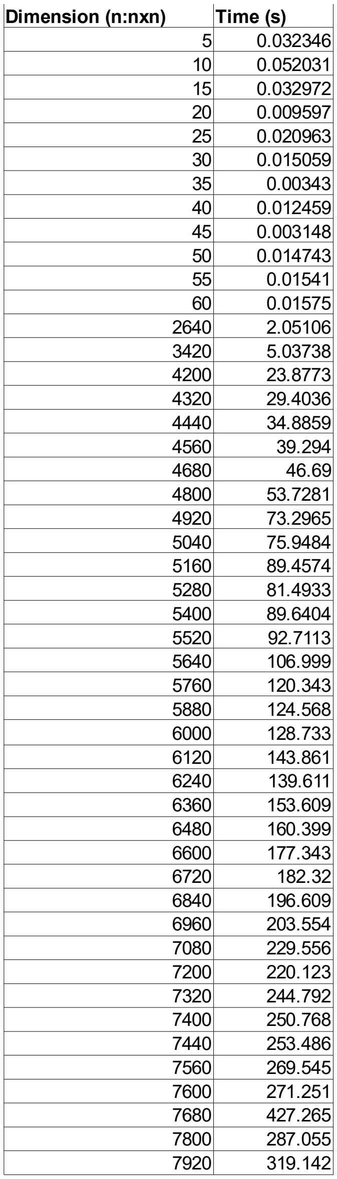


Figure 6: the run time of the parallel implementation using 25 cores.

The run times for this algorithm begin to increase sharply between 3000x3000 and 4000x4000 matrices. Notice that the peaks and valleys trend also occurs for 25 cores. The timing that jumps above the 400 second mark is not an example of this, but is caused by excessive communication time due to cluster traffic. Table 7 has the upper and lower bounds run time for matrices being multiplied using 25 cores.

Lower and Upper Bound Parallel Run Times on 25 Cores



Dimension (n:nxn)	Time (s)
5	0.032346
10	0.052031
15	0.032972
20	0.009597
25	0.020963
30	0.015059
35	0.00343
40	0.012459
45	0.003148
50	0.014743
55	0.01541
60	0.01575
2640	2.05106
3420	5.03738
4200	23.8773
4320	29.4036
4440	34.8859
4560	39.294
4680	46.69
4800	53.7281
4920	73.2965
5040	75.9484
5160	89.4574
5280	81.4933
5400	89.6404
5520	92.7113
5640	106.999
5760	120.343
5880	124.568
6000	128.733
6120	143.861
6240	139.611
6360	153.609
6480	160.399
6600	177.343
6720	182.32
6840	196.609
6960	203.554
7080	229.556
7200	220.123
7320	244.792
7400	250.768
7440	253.486
7560	269.545
7600	271.251
7680	427.265
7800	287.055
7920	319.142

Table 7:The lower and upper bounds run times for the parallel implementation using 25 cores.

When using 25 cores, matrices in the range of 7680x7680 to 7920x7920 were required to push the algorithm over the 5 minute mark. This is allows matrices with dimensions that are roughly twice as large as the matrix that pushed the sequential implementation to the 4 minute mark.

Speed Up:

The speed up was, in a word, super-linear. The greatest speed up experienced by the parallel implementation was around 140 when using 25 cores to multiply matrices of 2640x2640, which is over 5 times the expected maximum speed up amount. Cache, in a word, is the cause. When multiplying a 2640x2640 matrices using 25 cores, each processor stores 3 sub-matrices of 528x528. When looking at the sequential run times, it can be seen that the run time nearly doubles from the previous tested matrix size when multiplying 780x780 matrices. This is entirely a cache issue. Somewhere between multiplying matrices of 720x720 and matrices of 780x780 cache is no longer able to store all 3 matrices required in the multiplication. When multiplying 2640x2640 matrices using 25 cores, all 3 sub-matrices for each processor can likely fit in cache. This is why super-linear speed up is achieved by this algorithm. The author estimates that the lowest level cache size for each processor is somewhere around 6 MB based on the number of elements present in 3 matrices each with dimensions of 720x720. Figure 7 shows the true measured speed up for all configurations of cores tested.

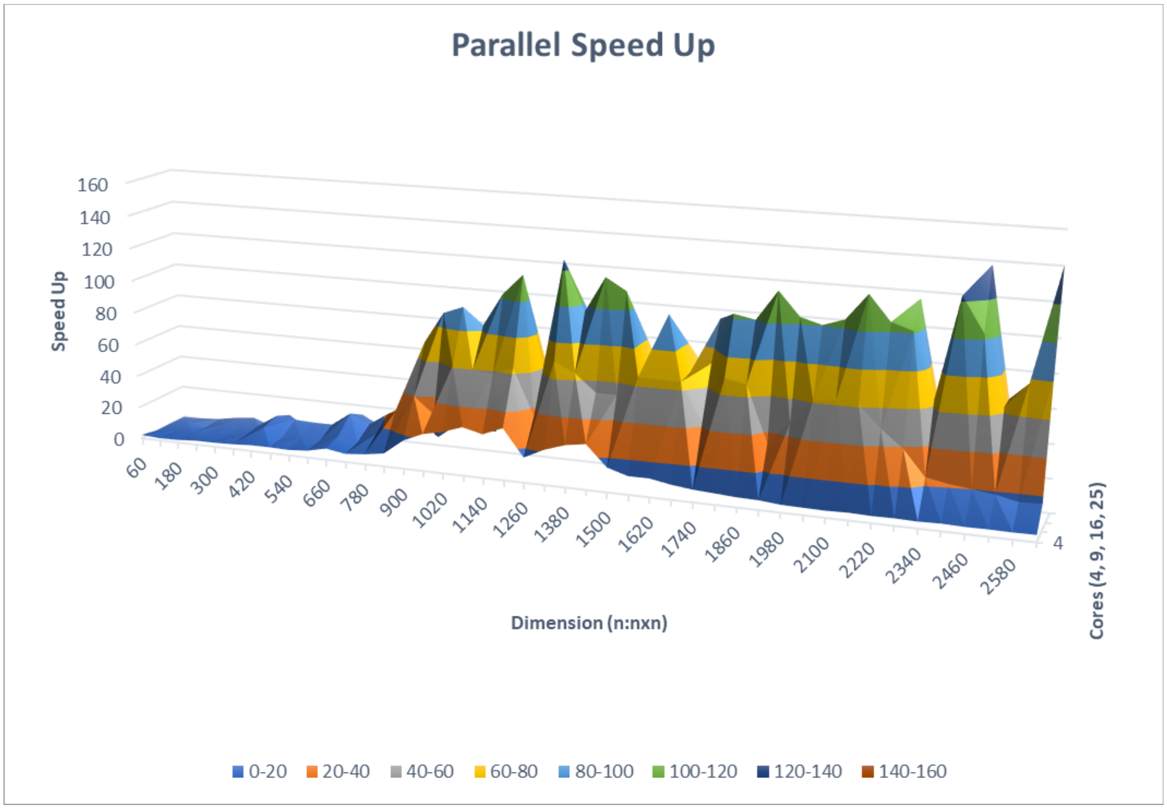


Figure 7: the speed up for all tested configurations of cores.

For the 16 core and 25 core configurations all 3 sub-matrices were able to fit in cache. Therefore, the graph shows a lot of impressive super-linear speed up. On larger matrices, however, the speed-up would become more reasonable. Table 8 shows the speed up data for Figure 7.

Speed Up 4, 9, 16, and 25 Cores

	Dimension (n:nxn)				
Cores		4	9	16	25
	60	2.1205357143	0.2733026467	0.3349788434	0.03015873
	120	1.3741658723	1.2700440529	1.4443887776	0.18984591
	180	1.609973425	2.0471079308	0.1072566703	1.0279469
	240	2.435589942	2.2075736325	3.3032528856	3.16275954
	300	2.4920116699	2.5593921667	4.4061655613	4.41321196
	360	2.8615292251	3.8863133372	5.3076678333	0.06615444
	420	3.6049368837	10.15678733	12.955384971	8.86022736
	480	3.6662311864	5.3091276629	11.789709945	0.95514691
	540	3.3704551558	6.198019802	11.226666667	0.6063212
	600	4.2318626243	5.641225435	11.239190748	1.46907559
	660	6.9633574224	10.126449929	19.621845015	14.2579955
	720	5.0060223216	7.3160044857	14.075592371	7.8414252
	780	6.1245204401	14.232197576	22.013150084	9.50127706
	840	8.0349683758	28.93677439	28.686283981	14.2473368
	900	17.31991311	40.110861835	69.173070084	5.03474946
	960	22.684984244	29.943899318	88.220040848	14.3535302
	1020	25.36218233	61.30967645	92.872594785	12.401091
	1080	29.532567645	59.977327517	83.011673113	12.6158676
	1140	26.558155352	58.550188606	102.82909752	37.1627926
	1200	31.578246321	63.144173884	115.21299469	30.012199
	1260	15.239673965	35.647814026	59.303508435	39.7362201
	1320	21.290207154	69.453019638	126.15947324	78.1744682
	1380	25.418928133	63.748681144	96.499257089	85.1775437
	1440	27.410619899	53.411558189	117.77844862	60.2147451
	1500	14.871304451	53.39726745	110.53791644	33.7631955
	1560	11.124589792	65.029148107	63.739250686	67.1574503
	1620	11.179737684	64.865550715	99.135704785	36.420687
	1680	8.9441743803	64.645843857	65.74860383	70.3427069
	1740	7.5004129672	41.564671493	72.402830377	94.7868024
	1800	6.7547321203	69.075997343	102.50721607	52.9887001
	1860	5.939346887	66.951644759	100.12007675	73.3789907
	1920	5.6784548556	33.862536075	117.91625897	47.646504
	1980	4.5258775393	46.833706038	103.86699614	94.7962745
	2040	4.2914871768	61.303981291	100.33989211	58.0735982
	2100	4.092674568	62.736127832	104.61058917	43.4066849
	2160	4.4368457226	61.748128473	120.12980022	66.7524088
	2220	4.0809054053	49.106392047	105.31845245	73.8049688
	2280	4.2530987246	38.981457872	101.24958949	115.913577
	2340	3.9546195245	24.010050765	63.439027687	56.2171005
	2400	4.326559556	21.715517135	123.2176322	81.5967472
	2460	3.9815600761	20.069690342	103.17901538	137.858655
	2520	4.1153858589	18.547894777	63.275023939	63.3848073
	2580	3.9760513778	15.698681579	77.733130309	71.8646134
	2640	4.1596856185	16.246740153	83.248096604	140.631186

Table 8: the speed up for 4, 9, 16, and 25 cores on matrix dimensions ranging from 60x60 to 2640x2640.

The greatest speed up achieved was 140.631186 by 25 cores multiplying 2640x2640 matrices. As expected, the majority of lower end (around 60x60 sized matrices) speed ups are around or below 1. This is caused by the communication time exceeding the save computation time. As the 16 core and 25 core runs are able to fit all of the matrices in the lowest level of cache, it is not surprising that both of these configurations have super-linear speed up for this entire range. The 4 core ans 9 core implementations each show an interesting cache phenomena. At the upper end, the 4 core thrashes between linear and super-linear speed up. This is likely caused by the upper levels of cache and higher instances of cache hits for certain sized matrices. The 9 core's super-linear speed up can be attributed to the fact that it uses two of cluster nodes and, in essence, gets an entire node's worth of extra cache. Figure 8 shows the individual speed ups of each configuration of cores up to their 5 minute mark.

Individual Speed Up Charts for each Core Configuration

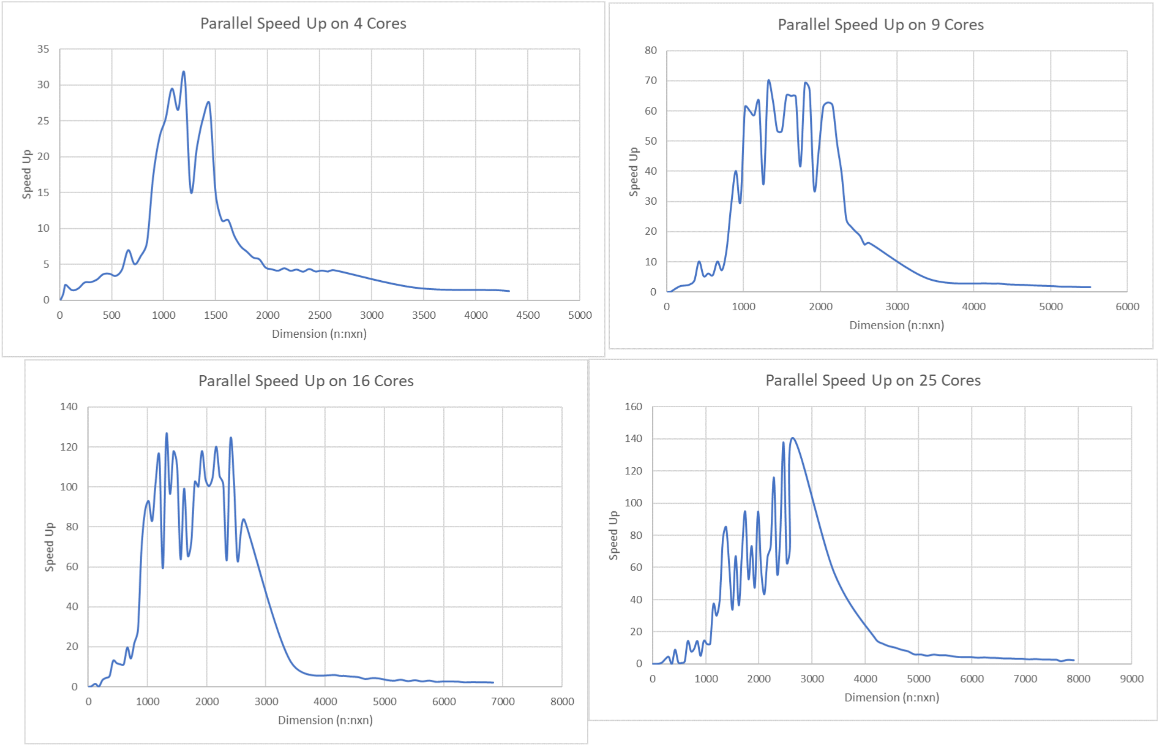
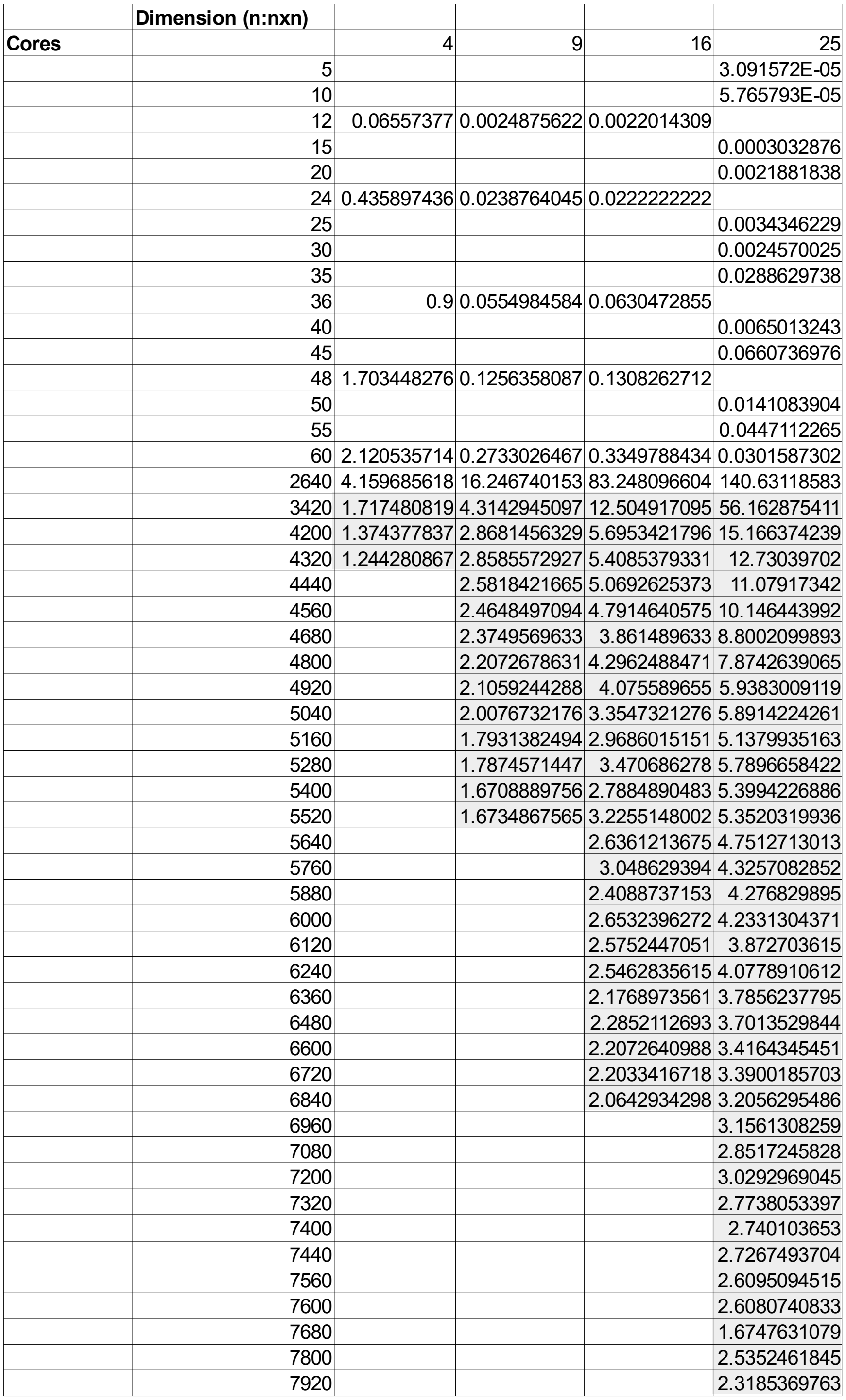


Figure 8: The speed up for each core configuration on a separate sub-plot.

Although the speed up is super-linear on smaller matrices, as the sub-matrices are no longer able to fit in cache, the speed up degrades to its expected linear levels for all configurations. Table 9 shows the speed up for the upper and lower range for each configuration.

Speed Up for the Upper and Lower Bounds for all Configurations



	Dimension (n:nxn)				
Cores		4	9	16	25
	5				3.091572E-05
	10				5.765793E-05
	12	0.06557377	0.0024875622	0.0022014309	
	15				0.0003032876
	20				0.0021881838
	24	0.435897436	0.0238764045	0.0222222222	
	25				0.0034346229
	30				0.0024570025
	35				0.0288629738
	36	0.9	0.0554984584	0.0630472855	
	40				0.0065013243
	45				0.0660736976
	48	1.703448276	0.1256358087	0.1308262712	
	50				0.0141083904
	55				0.0447112265
	60	2.120535714	0.2733026467	0.3349788434	0.0301587302
	2640	4.159685618	16.246740153	83.248096604	140.63118583
	3420	1.717480819	4.3142945097	12.504917095	56.162875411
	4200	1.374377837	2.8681456329	5.6953421796	15.166374239
	4320	1.244280867	2.8585572927	5.4085379331	12.73039702
	4440		2.5818421665	5.0692625373	11.07917342
	4560		2.4648497094	4.7914640575	10.146443992
	4680		2.3749569633	3.861489633	8.8002099893
	4800		2.2072678631	4.2962488471	7.8742639065
	4920		2.1059244288	4.075589655	5.9383009119
	5040		2.0076732176	3.3547321276	5.8914224261
	5160		1.7931382494	2.9686015151	5.1379935163
	5280		1.7874571447	3.470686278	5.7896658422
	5400		1.6708889756	2.7884890483	5.3994226886
	5520		1.6734867565	3.2255148002	5.3520319936
	5640			2.6361213675	4.7512713013
	5760			3.048629394	4.3257082852
	5880			2.4088737153	4.276829895
	6000			2.6532396272	4.2331304371
	6120			2.5752447051	3.872703615
	6240			2.5462835615	4.0778910612
	6360			2.1768973561	3.7856237795
	6480			2.2852112693	3.7013529844
	6600			2.2072640988	3.4164345451
	6720			2.2033416718	3.3900185703
	6840			2.0642934298	3.2056295486
	6960				3.1561308259
	7080				2.8517245828
	7200				3.0292969045
	7320				2.7738053397
	7400				2.740103653
	7440				2.7267493704
	7560				2.6095094515
	7600				2.6080740833
	7680				1.6747631079
	7800				2.5352461845
	7920				2.3185369763

Table 9: the speed up for upper and lower bounds for all tested core configurations. The cells in gray used sequential times that were predicted using simple linear regression.

As the table shows, as the dimensions of the matrices increase, all speed up becomes linear. This is caused by sub-matrices no longer fitting in cache combined with increasingly large messages being passed. By the time that matrices in the 3000x3000 to 4000x4000 dimensions range are reached, all super-linear speed up disappears. When multiplying matrices of 4200x4200 on 25 cores each processsor contains 3 sub-matrices of 840x840, which exceeds the size of the lowest level of cache that was found using the sequential algorithm. When the total data exceeds 6 MB per core, the expected linear or close to linear speed up occurs depending on whether or not an extra node's worth of cache is being used.

Efficiency:

Now that run time, speed up, and super-linear speed up have been explained it is time to discuss the efficiency of the parallel implementation. Given that efficiency is directly related to the speed up factor, the super-linear speed up caused by cache has to lead to efficiencies well above 100%. Figure 9 shows the efficiency of all tested core configurations as a surface chart.

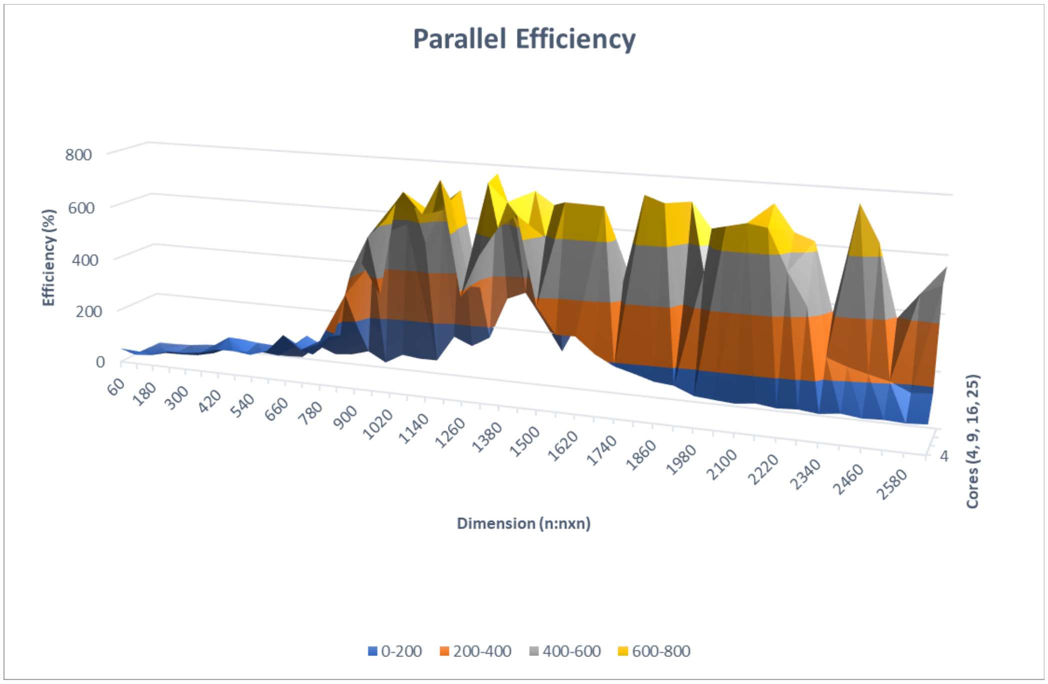
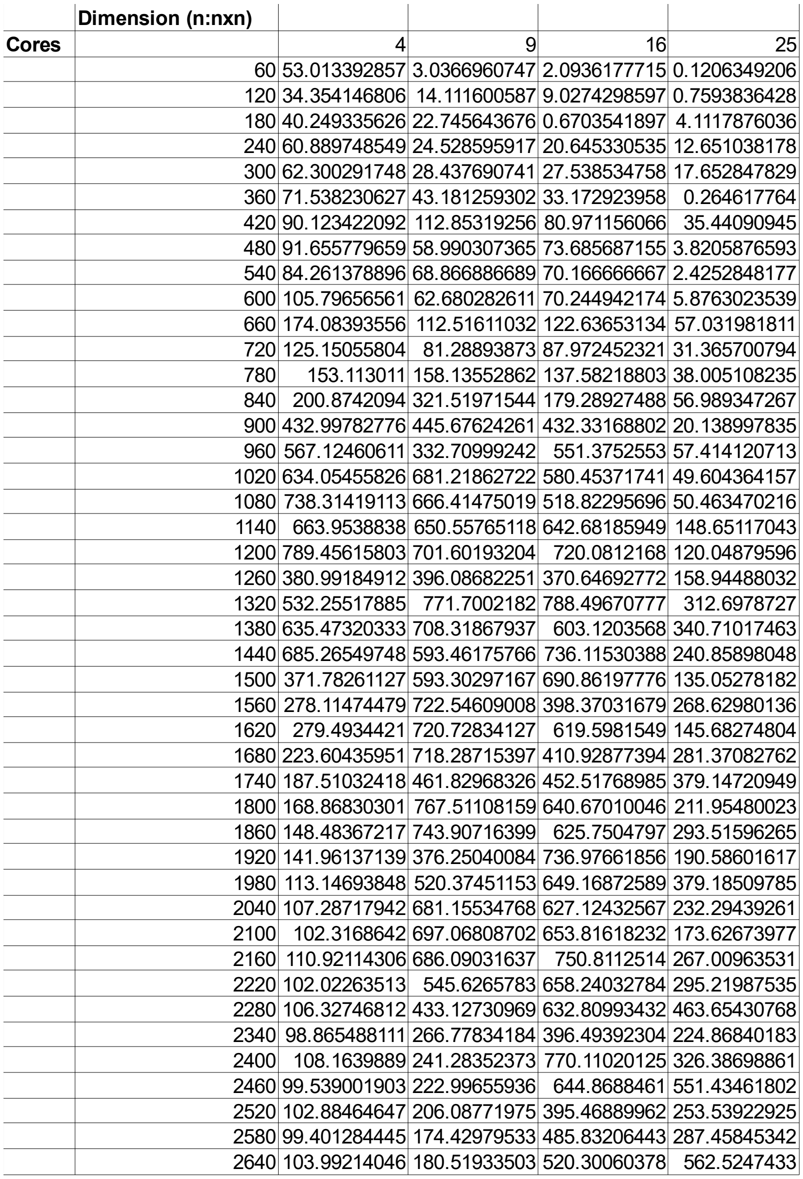


Figure 9: The efficiency of all tested configurations of cores on matrices of size 60x60 to 2640x2640. Due to super-linear speed up, the efficiency tops out in the 600% to 800% range.

The efficiency achieved by this implementation of Cannon's Algorithm is impressively high. For matrices of 1380x1380 all of the configurations achieved efficiencies above 200%. This can be attributed to the huge speed up factor that is enabled by all 3 sub-matrices at each processor fitting entirely in cache. Table 10 shows the efficiency values for the implementation used to create Figure 9.

Efficiency of All Tested Core Configurations



	Dimension (n:nxn)				
Cores		4	9	16	25
	60	53.013392857	3.0366960747	2.0936177715	0.1206349206
	120	34.354146806	14.111600587	9.0274298597	0.7593836428
	180	40.249335626	22.745643676	0.6703541897	4.1117876036
	240	60.889748549	24.528595917	20.645330535	12.651038178
	300	62.300291748	28.437690741	27.538534758	17.652847829
	360	71.538230627	43.181259302	33.172923958	0.264617764
	420	90.123422092	112.85319256	80.971156066	35.44090945
	480	91.655779659	58.990307365	73.685687155	3.8205876593
	540	84.261378896	68.866886689	70.166666667	2.4252848177
	600	105.79656561	62.680282611	70.244942174	5.8763023539
	660	174.08393556	112.51611032	122.63653134	57.031981811
	720	125.15055804	81.28893873	87.972452321	31.365700794
	780	153.113011	158.13552862	137.58218803	38.005108235
	840	200.8742094	321.51971544	179.28927488	56.989347267
	900	432.99782776	445.67624261	432.33168802	20.138997835
	960	567.12460611	332.70999242	551.3752553	57.414120713
	1020	634.05455826	681.21862722	580.45371741	49.604364157
	1080	738.31419113	666.41475019	518.82295696	50.463470216
	1140	663.9538838	650.55765118	642.68185949	148.65117043
	1200	789.45615803	701.60193204	720.0812168	120.04879596
	1260	380.99184912	396.08682251	370.64692772	158.94488032
	1320	532.25517885	771.7002182	788.49670777	312.6978727
	1380	635.47320333	708.31867937	603.1203568	340.71017463
	1440	685.26549748	593.46175766	736.11530388	240.85898048
	1500	371.78261127	593.30297167	690.86197776	135.05278182
	1560	278.11474479	722.54609008	398.37031679	268.62980136
	1620	279.4934421	720.72834127	619.5981549	145.68274804
	1680	223.60435951	718.28715397	410.92877394	281.37082762
	1740	187.51032418	461.82968326	452.51768985	379.14720949
	1800	168.86830301	767.51108159	640.67010046	211.95480023
	1860	148.48367217	743.90716399	625.7504797	293.51596265
	1920	141.96137139	376.25040084	736.97661856	190.58601617
	1980	113.14693848	520.37451153	649.16872589	379.18509785
	2040	107.28717942	681.15534768	627.12432567	232.29439261
	2100	102.3168642	697.06808702	653.81618232	173.62673977
	2160	110.92114306	686.09031637	750.8112514	267.00963531
	2220	102.02263513	545.6265783	658.24032784	295.21987535
	2280	106.32746812	433.12730969	632.80993432	463.65430768
	2340	98.865488111	266.77834184	396.49392304	224.86840183
	2400	108.1639889	241.28352373	770.11020125	326.38698861
	2460	99.539001903	222.99655936	644.8688461	551.43461802
	2520	102.88464647	206.08771975	395.46889962	253.53922925
	2580	99.401284445	174.42979533	485.83206443	287.45845342
	2640	103.99214046	180.51933503	520.30060378	562.5247433

Table 10: the efficiency (%) of all tested core configurations on matrices of size 60x60 to 2640xx2640.

The best efficiencies were well above 700% and the worst efficiencies were below 1%. The worst efficiencies occurred in instances were a large number of cores were used to multiply small matrices. This caused the communication time to exceed the time saved by the division of computations between cores. Figure 10 shows individual plots that of the efficiency for each tested core configuration.

Individual Plots of Efficiency for 4, 9, 16, and 25 Cores

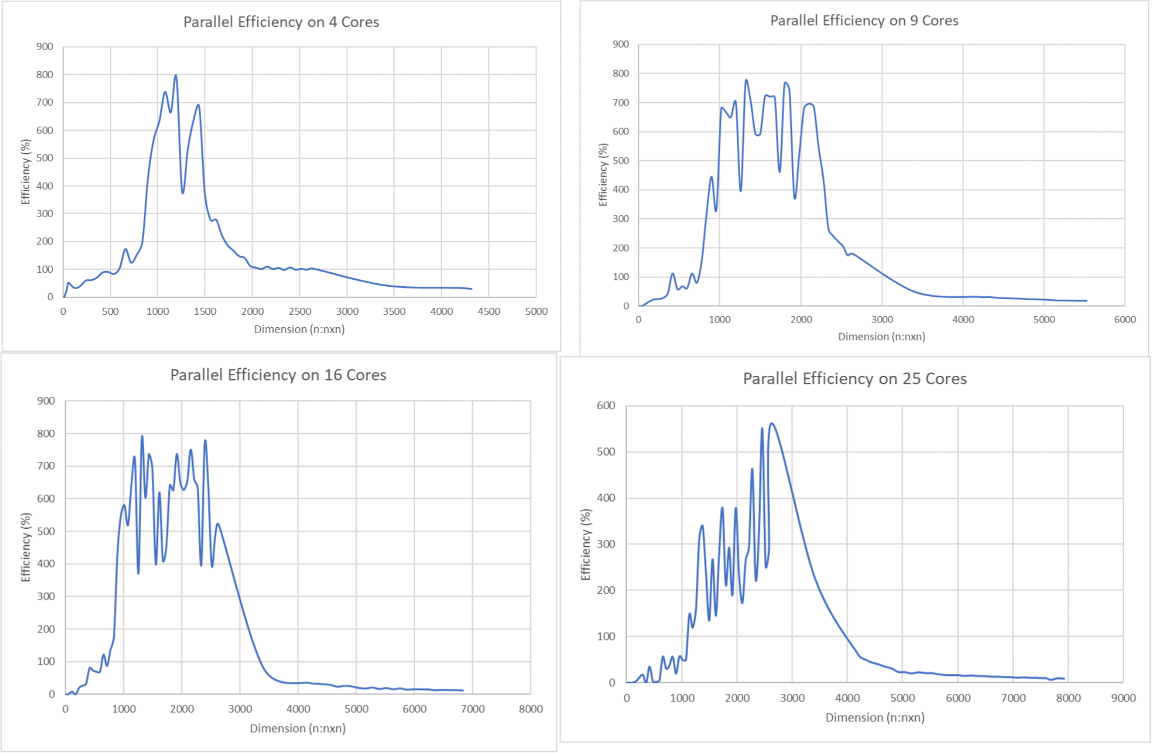
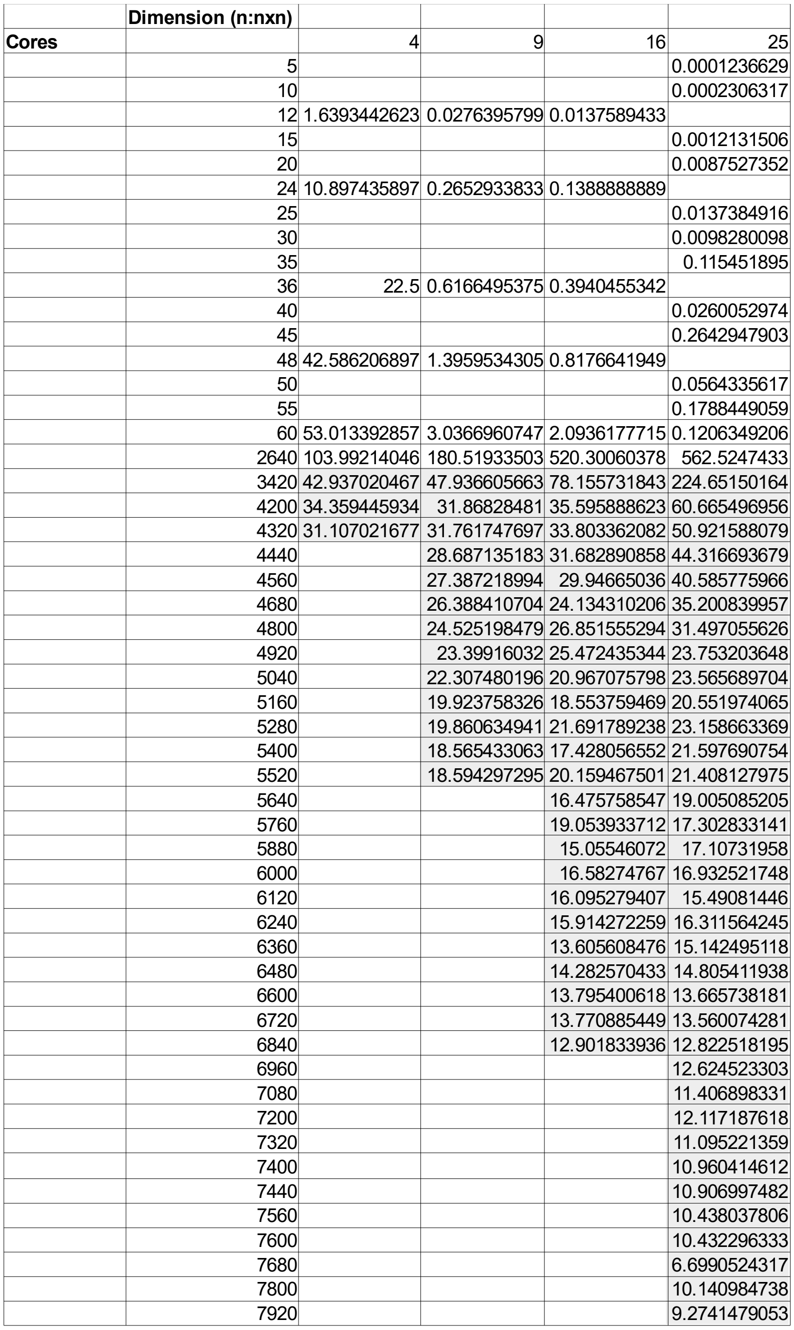


Figure 10: individual plots of the efficiency for 4, 9, 16, and 25 cores. Each plot shows the efficiency until the run time for the number of cores corresponding to that plot reached or exceeded 5 minutes.

As the individual plots for each core configuration demonstrate, the efficiency, like speed up, drops below 100% as the size of the 3 sub-matrices exceeds the cache size. Since efficiency is essentially a speed up factor scaled by the inverse of the number of cores, the efficiency plots look almost the same as the speed up plots. Table 11 shows the efficiencies of the upper and lower bounds of matrix dimensions tested for each core configuration.

Efficiency for the Upper and Lower Bounds for all Configurations



	Dimension (n:nxn)				
Cores		4	9	16	25
	5				0.0001236629
	10				0.0002306317
	12	1.6393442623	0.0276395799	0.0137589433	
	15				0.0012131506
	20				0.0087527352
	24	10.897435897	0.2652933833	0.1388888889	
	25				0.0137384916
	30				0.0098280098
	35				0.115451895
	36	22.5	0.6166495375	0.3940455342	
	40				0.0260052974
	45				0.2642947903
	48	42.586206897	1.3959534305	0.8176641949	
	50				0.0564335617
	55				0.1788449059
	60	53.013392857	3.0366960747	2.0936177715	0.1206349206
	2640	103.99214046	180.51933503	520.30060378	562.5247433
	3420	42.937020467	47.936605663	78.155731843	224.65150164
	4200	34.359445934	31.86828481	35.595888623	60.665496956
	4320	31.107021677	31.761747697	33.803362082	50.921588079
	4440		28.687135183	31.682890858	44.316693679
	4560		27.387218994	29.94665036	40.585775966
	4680		26.388410704	24.134310206	35.200839957
	4800		24.525198479	26.851555294	31.497055626
	4920		23.39916032	25.472435344	23.753203648
	5040		22.307480196	20.967075798	23.565689704
	5160		19.923758326	18.553759469	20.551974065
	5280		19.860634941	21.691789238	23.158663369
	5400		18.565433063	17.428056552	21.597690754
	5520		18.594297295	20.159467501	21.408127975
	5640			16.475758547	19.005085205
	5760			19.053933712	17.302833141
	5880			15.05546072	17.10731958
	6000			16.58274767	16.932521748
	6120			16.095279407	15.49081446
	6240			15.914272259	16.311564245
	6360			13.605608476	15.142495118
	6480			14.282570433	14.805411938
	6600			13.795400618	13.665738181
	6720			13.770885449	13.560074281
	6840			12.901833936	12.822518195
	6960				12.624523303
	7080				11.406898331
	7200				12.117187618
	7320				11.095221359
	7400				10.960414612
	7440				10.906997482
	7560				10.438037806
	7600				10.432296333
	7680				6.6990524317
	7800				10.140984738
	7920				9.2741479053

Table 9: the efficiency (%) for upper and lower bounds for all tested core configurations. The cells in gray used sequential times that were predicted using simple linear regression.

Just like speed up, as the 3 sub-matrices no longer fit in cache memory for each core the efficiency drops below 100%. Based on the predicted sequential times, the efficiency of the parallel algorithm will approach 0% as the size of the matrices being multiplied increase. This can be attributed to increasingly expensive communications as larger and larger buffers are swapped. Cache enabled a range of matrix dimensions on a range of square core sizes that achieved super-linear speed up and efficiencies above 100%.

Conclusion:

In conclusion, Cannon’s Algorithm provides an excellent method to multiply matrices over a mesh of processors. The algorithm not only divides the labor among all cores equally, but also allows for the better utilization of cache memory that creates super-linear speed-up. Based on data collected for the sequential algorithm, the author believes that the lowest level of cache is around 6 MB or 3 matrices of 720x720. Through the division of the problem into sub-matrices across processors Cannon’s Algorithm is able to use the 6 MB cache size to achieve super-linear speed up on comparatively large matrices. Indeed for most matrices less than 4200x4200 being multiplied by 25 cores, all 3 sub-matrices can fit in cache, thus allowing for super-linear speed up and efficiencies above 100%.