Report on Project 4, Part 5

Parallelization of Matrix Multiplication

CS415

May 3, 2017

Evan Su

Version 5.0

Table of Contents

Overview:

Computing matrix multiplication is known to have a long calculation time. Matrix multiplication itself is not necessarily difficult but as the size of the matrix grows, more calculations are needed. Computing matrix multiplication has a run time of n^3. The purpose of this experiment is to show the speedup achieved with parallelization.

Test Methodology:

Two different programs are written to test the matrix multiplication time, sequential and parallel. The sequential code will be used as a controlled variable. The code will be tested at various sizes starting at 360 by 360 sized matrix. For more detail of the sequential and parallel code, please see below.

Sequential

The sequential code first allocates spaces for three matrixes, 2 for multiplicand and 1 for product. The two multiplicand matrixes are filled. After being filled, the two matrixes are multiplied together and the results are stored in the product matrix. The time starts when the calculation starts and ends when the calculation ends.

Parallel

The parallel code first allocates spaces for three tiles in all the cores, 2 for multiplicand and 1 for product. The size of the tiles is determined by the size of the matrix and the number of cores. The two multiplicand matrixes are filled. After being filled, all the cores will follow Cannon's algorithm to multiple the two matrixes (spread across all the tiles) together. The time starts when the calculation starts and ends when the calculation ends.

Code Revisions:

Addressing Comments:

I have received the following concerns over my initial parallel implementation and addressed them accordingly. I grouped similar concerns together so that I can address them all at once.

- Code is very dense. Spacing out code statements with new line would make is easier to read.
- Code is very dense and bit difficult to read. This is partly due to it being written in C but there is a lack of auxiliary functions to make the code much more readable. Also line spacing would be nice to see.

  Resulting Changes:
  I added more line space between different lines of code. I also added additional functions for complex row and column id calculations. I added section headers for long functions to add readability.

- The code would also benefit from some more functions to make the code easier to read. For instance, a allocateMatrix function would be nice to see. It would also be nice to see a functions for sending and receiving.
- Although the functionality could be split into subroutines a little more, the overall code structure is solid.
- Everything was nice and understandable. I might put some of the repeated things in a function(matrix memory allocation, for example).

  Resulting Changes:
  I added an allocate matrix function, free matrix function and calculate neighbor function to reduce the amount of repeating code. I did not added send and receive functions because the function would only be used once at the beginning with a very specific set of circumstances to work.

- Instead of the switch statement it would nice to see just the sqrt() function in math.h and then some checking. It is possible to run this code on the cluster with 49 cores.
- On a more style-related note, you could make your switch statement at the beginning check for a remainder when you divide the size by it's square root, which would make it a bit less verbose and help it handle the general case.

  Resulting Changes:
  I change the switch statement to a square root function

- I really like the print flag.

  Resulting Changes:
  I added more flags for printing matrix and printing time

- I'm not sure what the output time means. I'm assuming its in micro seconds. That would be nice to see in the documentation.

  Resulting Changes:
  I added additional notes in the documentation explicitly stating that the time printout is in microseconds. I did not change the printout format because I wanted to keep the output in csv format

- The only markdown is mostly from not collecting the final result into one matrix.

  Resulting Changes:
  I added a function that collects all the data from the tiles and puts them into one matrix

- Not that it's super important, but you might clean this[makefile] up a bit(there's still stuff that looks like it's from PA1). It's really just for your own benefit, but it might help if you have to debug something in you makefile.

  Resulting Changes:
  I glad this person noticed the relics from previous projects in my makefile. I am not making any immediate changes to the makefile in case of the off chance of needing to use old code again.

- You might speed up your initial matrix communication time by transposing things to begin with to avoid the whole doubly-nested-for-loop inside a triply-nested-for-loop thing.

  Resulting Changes:
  I did not change my matrix communication method because I do not possess the knowledge to implement the feature of transposing matrices.

Data Analysis:

The raw data from the tests can be found in the file project4Analysis.

Sequential

The run time of the sequential code exhibits a runtime of O(n^3) as shown in Graph 1.1. The execution time grows exponentially as the length of the matrix grows linearly.

Parallel

The parallel implementation of matrix multiplication showed a decrease of overall runtime compared to the sequential implementation. The speedup calculation indicated that there was super linear speed up as the values were significantly greater than the cores used as shown in graph 2.1, graph 2.2 and graph 3.1. The super linear speedup did not show up in larger matrix sizes as shown in graph 5.1, and graph 5.2. The efficiency appears to decrease as the size of the matrix grows. The decrease is due to the increase communication time as the cores had to send larger matrices to each other. The decreases can be seen in graph 5.3. The various dips and peaks in these graphs can be attributed to the number of users on the cluster. The communication time increased as more users ran their application on the shared cluster.

Increasing the number of cores also increases the largest matrix that can be calculated within a given amount of time as shown on graph 4.1. The matrix size appears to grow logarithmically as the number of cores increase. These results indicate that increasing the number of cores would be give diminishing returns on very large matrices (beyond 10,000 matrix length). The sudden jump length jump from 4 to 9 cores is due to the introduction of another box. The 9^{th} core utilize all the resources of an additional box.

Super Linear Speedup Explanation

The super linear speedup is likely due to the core's cache. On larger matrixes, the sequential implementation had more cache misses which resulted in more memory fetch time. The parallel implementation used the cache resource of multiple cores instead of one.

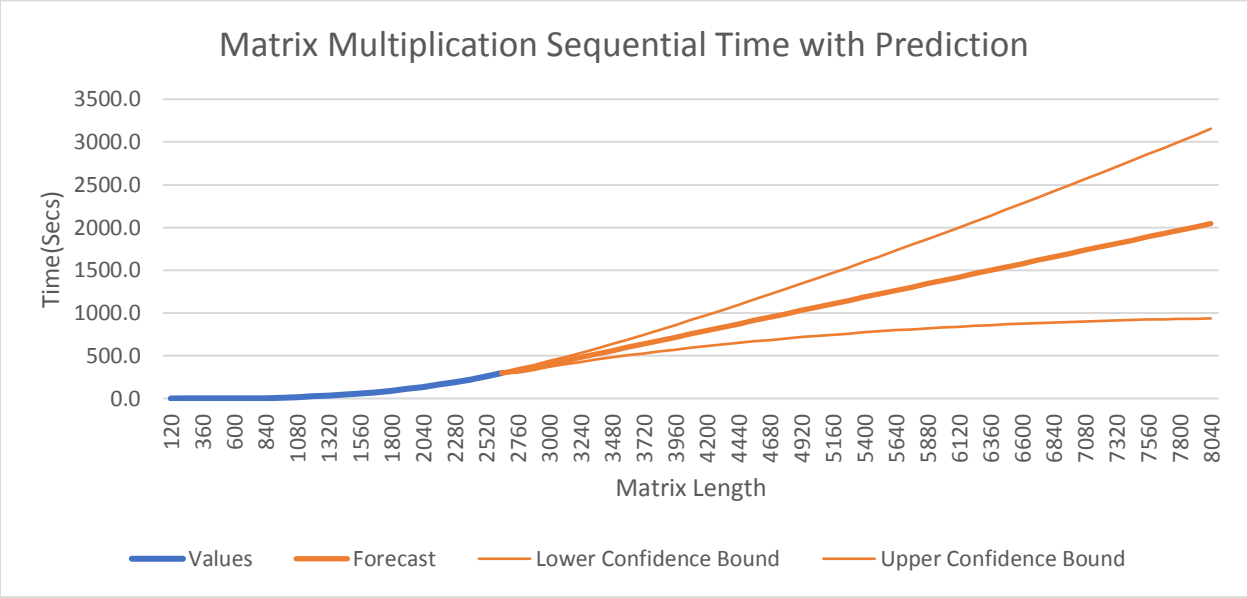
Conclusion:

Matrix multiplication has a runtime of O(n^3). Adding more cores would decrease the overall runtime. For larger matrices, each addition core would give diminishing returns on both speedup and efficiency. Thus, a different parallel matrix multiplication algorithm should be developed so that communication time is less.

Parallelization of application appears to benefit more from higher memory than higher clock speeds. The cache of the CPU was major influence on the runtime of matrix multiplication. High performance computers should focus on improving the memory size and access time.

Graphs and Tables

Graph 1.1



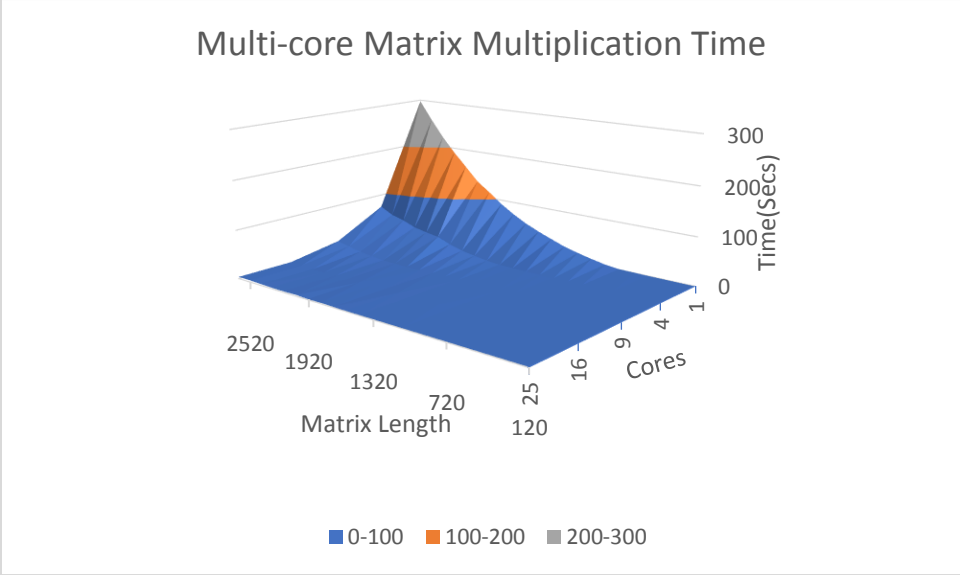
The graph shows the runtime of matrix multiplication is O(n^3). The orange line is a prediction of the runtime or matrix multiplication. The predicted data was produced by Microsoft Excel's Forecast function. The upper bound was used as a control for this experiment.

Table 1.1

Sequential Matrix runtime(secs)				
Matrix Length	Time(secs)	Forecast	Lower Confidence Bound	Upper Confidence Bound
120	0.005			
240	0.024			
360	0.136			
480	0.210			
600	0.600			
720	0.895			
840	4.214			
960	10.076			
1080	16.743			
1200	24.388			
1320	34.929			
1440	45.805			
1560	58.154			
1680	73.324			
1800	91.174			
1920	112.603			
2040	132.670			
2160	161.636			
2280	189.952			
2400	221.616			
2520	258.056			
2640	297.454	297.454	297.454	297.454
2760		323.957	298.827	349.086
2880		363.093	335.009	391.178
3000		402.230	368.435	436.024
3120		441.366	399.297	483.435
3240		480.503	428.061	532.945
3360		519.640	455.129	584.150
3480		558.776	480.787	636.766
3600		597.913	505.227	690.599
3720		637.049	528.583	745.516
3840		676.186	550.952	801.420
3960		715.323	572.408	858.237
4080		754.459	593.008	915.911
4200		793.596	612.799	974.393
4320		832.732	631.819	1033.646

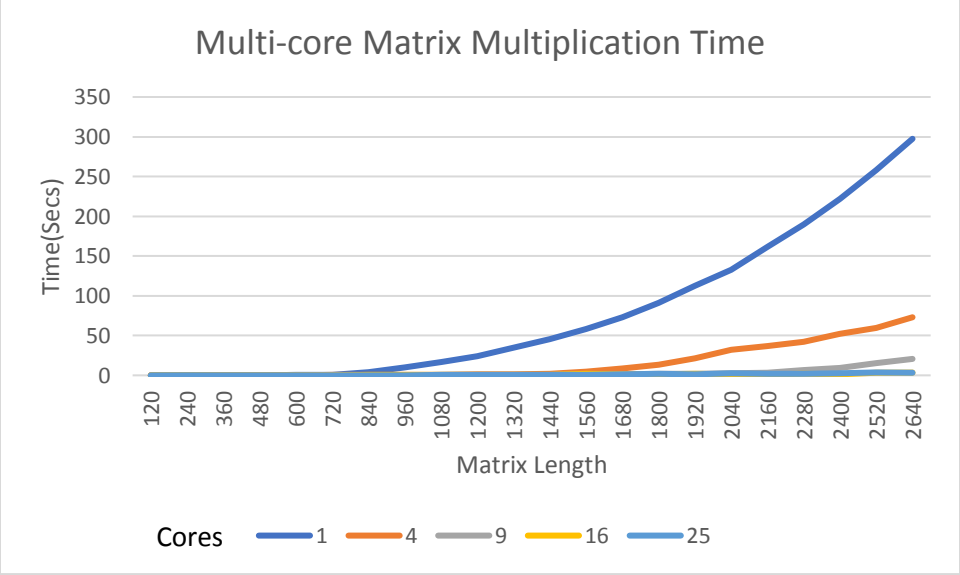
The table is the average sequential runtime of matrix multiplication. The runtime is O(n^3) as the length increases, the runtime increases exponentially. The predicted data was produced by Microsoft Excel's Forecast function. The table shown only goes to 4320 matrix length, the remaining table can be found in the file Project4Analysis.

Graph 1.2



The graph indicates that as the size of the matrix increases, the run time increases exponentially, O(n^3). As the number of cores increases, the run time decreases.

Graph1.3



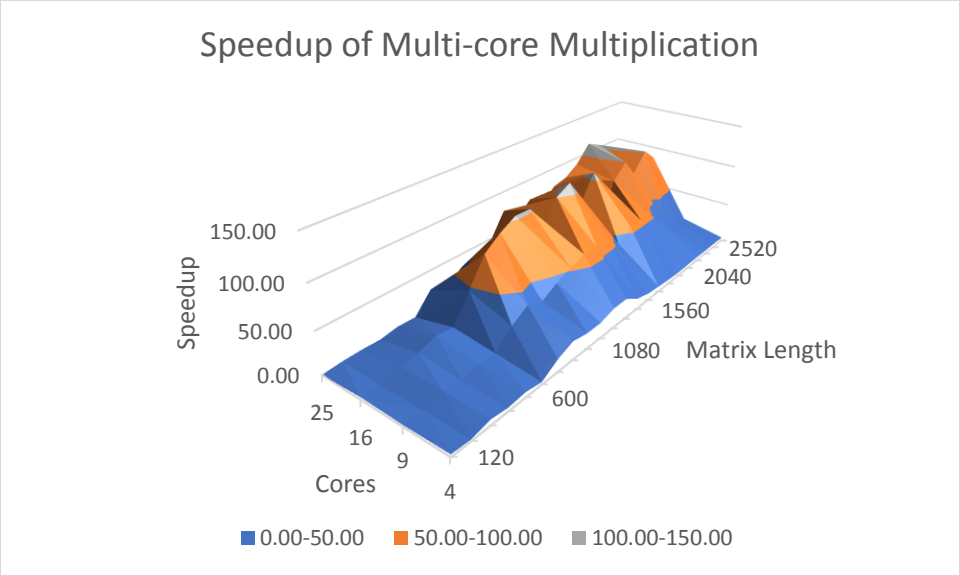
The graph indicates that as the size of the matrix increases, the run time increases exponentially, O(n^3). As the number of cores increases, the run time decreases.

Table 1.2

Runtime of Matrix Multiplication for Multi-cores						
		Cores				
	Time(Secs)	1	4	9	16	25
Matrix Length	120	0.005	0.002	0.003	0.016	0.080
	240	0.024	0.013	0.010	0.015	0.009
	360	0.136	0.020	0.015	0.227	0.016
	480	0.210	0.049	0.036	0.067	0.221
	600	0.600	0.097	0.060	0.228	0.080
	720	0.895	0.290	0.097	0.057	0.254
	840	4.214	0.276	0.148	0.276	0.082
	960	10.076	0.445	0.247	0.536	0.152
	1080	16.743	0.883	0.493	0.343	0.434
	1200	24.388	1.280	0.548	0.672	0.639
	1320	34.929	1.435	0.672	0.668	0.509
	1440	45.805	1.869	0.789	0.872	0.348
	1560	58.154	4.361	0.997	1.748	0.762
	1680	73.324	8.671	1.279	1.124	1.466
	1800	91.174	13.582	2.161	1.536	1.674
	1920	112.603	21.580	2.055	1.946	1.279
	2040	132.670	32.273	2.448	1.381	2.501
	2160	161.636	36.597	3.050	1.503	1.859
	2280	189.952	42.152	6.322	1.744	1.869
	2400	221.616	52.298	9.564	1.500	2.623
	2520	258.056	59.432	15.336	2.986	3.659
	2640	297.454	73.021	20.662	3.400	3.515

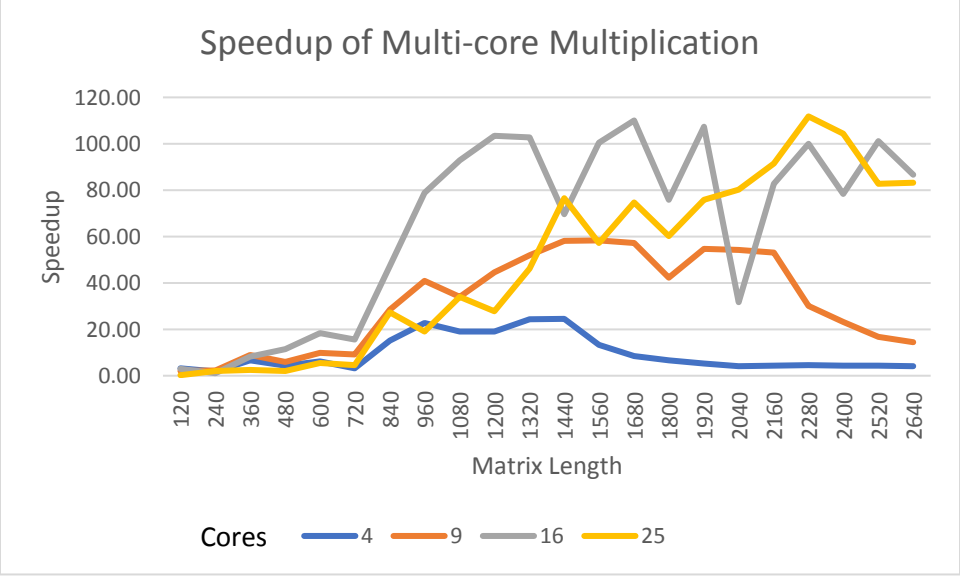
The table shows that as the size of the matrix increases, the run time increases exponentially, O(n^3). As the number of cores increases, the run time decreases.

Graph 2.1



The speedup increased as the matrix length increases. The speedup of matrix multiplication had super linear speedup due to cache hits and misses. Before 720 matrix length, the speedup was caused by the cannons algorithm. The valleys and peeks are caused by variations in communication time.

Graph 2.2



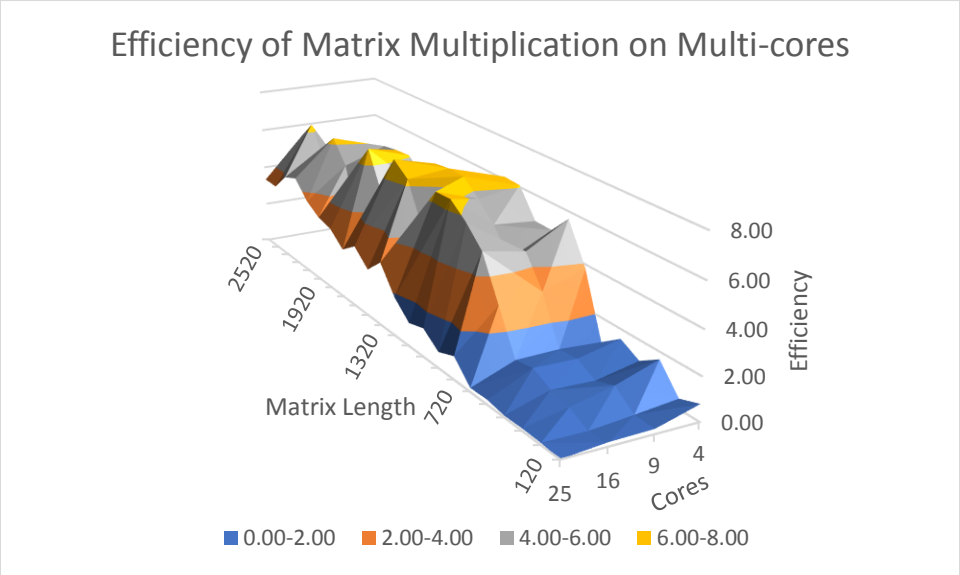
The speedup increased as the matrix length increases. The speedup of matrix multiplication had super linear speedup due to cache hits and misses. Before 720 matrix length, the speedup was caused by the cannons algorithm. The valleys and peeks are caused by variations in communication time.

Table 2.1

Speedup of Matrix Multiplication					
		Cores			
	Time	4	9	16	25
Matrix Length	120	3.174288	2.040878	3.010622	0.226268
	240	1.832708	2.246434	1.063859	2.130131
	360	6.643499	8.84543	8.27952	2.588517
	480	4.257068	5.832976	11.39994	1.967436
	600	6.218285	9.926148	18.32342	5.422681
	720	3.082152	9.190544	15.61688	4.496392
	840	15.25025	28.39201	47.25857	27.30588
	960	22.65008	40.87412	78.81778	18.96968
	1080	18.97142	33.93533	92.87894	33.94777
	1200	19.05871	44.49731	103.3761	27.84595
	1320	24.33522	51.95733	102.8519	46.10939
	1440	24.51311	58.06565	69.60044	76.48202
	1560	13.33416	58.33895	100.5103	57.11394
	1680	8.45598	57.32458	110.0441	74.61445
	1800	6.712725	42.18306	75.7669	60.2039
	1920	5.21802	54.78922	107.3852	75.9508
	2040	4.110856	54.19438	31.78422	80.30132
	2160	4.416694	52.98842	82.83528	91.47068
	2280	4.506363	30.04519	100.0879	111.7905
	2400	4.237543	23.17299	78.37005	104.2451
	2520	4.342043	16.82695	101.2066	82.83286
	2640	4.073531	14.39612	86.70098	83.24022

The speedup increased as the matrix length increases. The speedup of matrix multiplication had super linear speedup due to cache hits and misses. Before 720 matrix length, the speedup was caused by the cannons algorithm. After 2520 matrix length, the speedup began to decrease for 4 cores, 9 cores, 16 cores.

Graph 3.1



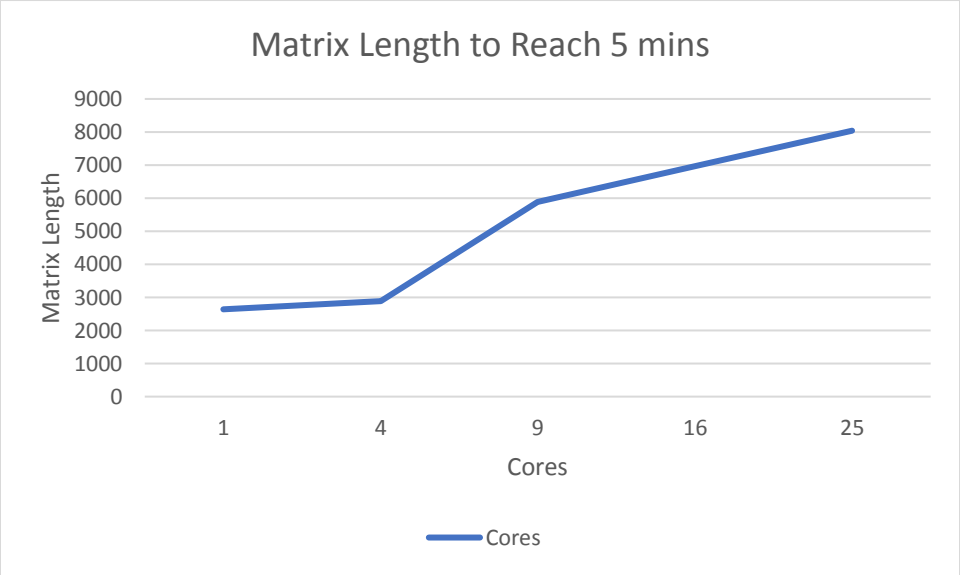
The efficiency increased as the matrix length increases. The efficiency of matrix multiplication was beyond 100% at certain points due to cache hits and misses. The valleys and peeks are caused by variations in communication time.

Table 3.2

Efficiency of Matrix Multiplication					
		Cores			
	Time	4	9	16	25
Matrix Length	120	0.793572	0.226764	0.188164	0.009051
	240	0.458177	0.249604	0.066491	0.085205
	360	1.660875	0.982826	0.51747	0.103541
	480	1.064267	0.648108	0.712497	0.078697
	600	1.554571	1.102905	1.145213	0.216907
	720	0.770538	1.021172	0.976055	0.179856
	840	3.812563	3.154668	2.95366	1.092235
	960	5.662521	4.541569	4.926111	0.758787
	1080	4.742854	3.770593	5.804933	1.357911
	1200	4.764678	4.944146	6.461003	1.113838
	1320	6.083806	5.773036	6.428242	1.844375
	1440	6.128278	6.451739	4.350027	3.059281
	1560	3.333541	6.482106	6.281895	2.284558
	1680	2.113995	6.369398	6.877756	2.984578
	1800	1.678181	4.687006	4.735431	2.408156
	1920	1.304505	6.087691	6.711575	3.038032
	2040	1.027714	6.021597	1.986514	3.212053
	2160	1.104173	5.887603	5.177205	3.658827
	2280	1.126591	3.338354	6.255494	4.471621
	2400	1.059386	2.574776	4.898128	4.169805
	2520	1.085511	1.869661	6.325412	3.313315
	2640	1.018383	1.599569	5.418811	3.329609

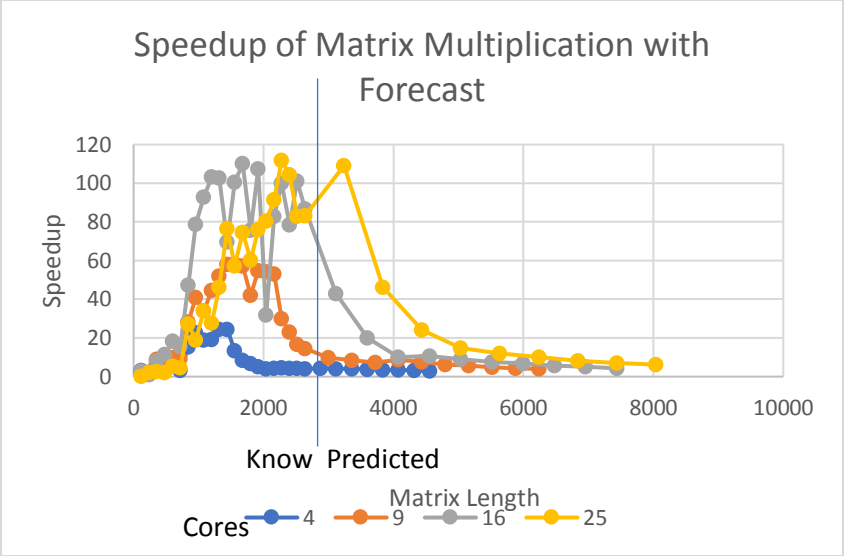
The efficiency increased as the matrix length increases. The efficiency of matrix multiplication was beyond 100% at certain points due to cache hits and misses.

Graph 4.1



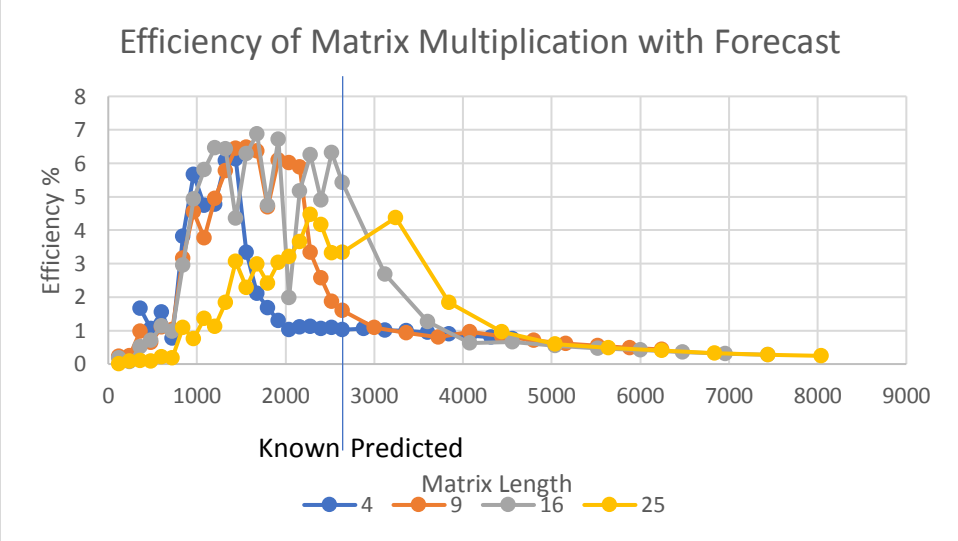
Given a finite amount of time, the matrix sized increased logarithmically as the number of cores increased. The spike from 4 cores to 9 cores is caused by the additional resources of another box.

Graph 5.1



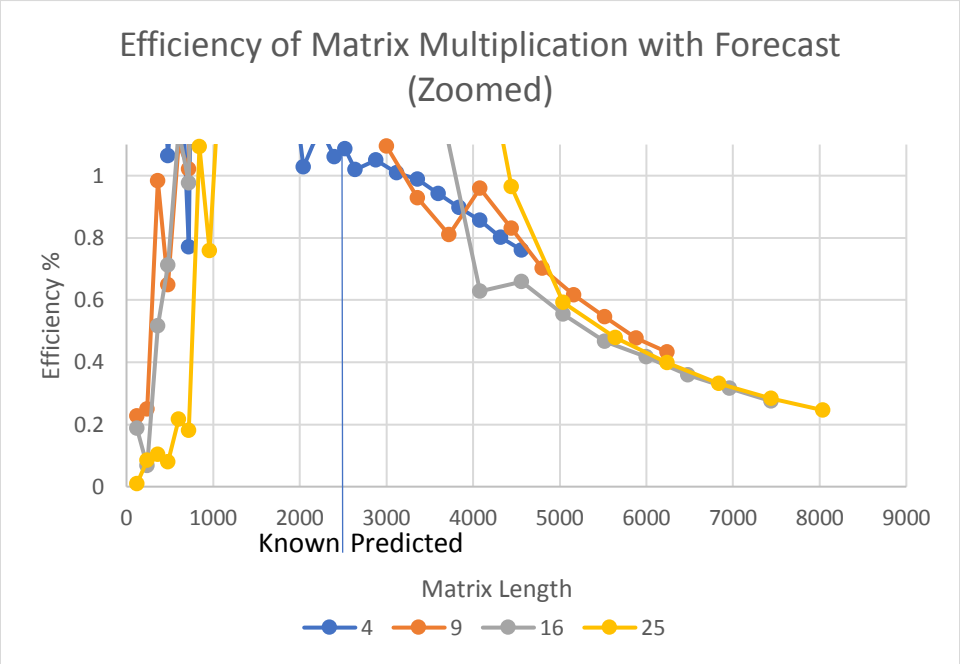
The speedup increased as the matrix length increases. The speedup of matrix multiplication had super linear speedup due to cache hits and misses. For very large matrixes, the speedup dropped significantly as all the cores were experiencing cache misses. The valleys and peeks are caused by variations in communication time.

Graph 5.2



The efficiency increased as the matrix length increases. The efficiency of matrix multiplication was beyond 100% at certain points due to cache hits and misses. For very large matrixes, the efficiency dropped significantly as all the cores were experiencing cache misses. The valleys and peeks are caused by variations in communication time.

Graph 5.3



The efficiency increased as the matrix length increases. The efficiency of matrix multiplication was beyond 100% at certain points due to cache hits and misses. For very large matrixes, the efficiency dropped significantly as all the cores were experiencing cache misses. The efficiency dropped below 100% for large matrices which implies that the parallel implementation was not able to utilize the cache similar to smaller matrices. The valleys and peeks are caused by variations in communication time.

Table 6.1

4 Core Statistics			
Matrix Length	Runtime	speed up	Efficency
120	1714.40	3.17	0.79
240	12845.20	1.83	0.46
360	20488.00	6.64	1.66
480	49438.60	4.26	1.06
600	96559.20	6.22	1.55
720	290318.80	3.08	0.77
840	276345.60	15.25	3.81
960	444873.00	22.65	5.66
1080	882529.80	18.97	4.74
1200	1279637.80	19.06	4.76
1320	1435332.40	24.34	6.08
1440	1868596.20	24.51	6.13
1560	4361262.00	13.33	3.33
1680	8671224.20	8.46	2.11
1800	13582222.00	6.71	1.68
1920	21579591.20	5.22	1.30
2040	32273045.20	4.11	1.03
2160	36596535.20	4.42	1.10
2280	42151995.20	4.51	1.13
2400	52298293.60	4.24	1.06
2520	59431979.20	4.34	1.09
2640	73021131.20	4.07	1.02
2880	93091460.00	4.20	1.05
3120	119713224.00	4.04	1.01
3360	147727968.00	3.95	0.99
3600	183223416.00	3.77	0.94
3840	223378460.00	3.59	0.90
4080	267472844.00	3.42	0.86
4320	322507640.00	3.21	0.80
4560	379355592.00	3.04	0.76

Raw statistical data for 4 core matrix multiplication

Table 6.2

9 Core Statistics			
Matrix Length	Runtime	speed up	Efficency
120	2666.50	2.04	0.23
240	10479.50	2.25	0.25
360	15387.83	8.85	0.98
480	36081.67	5.83	0.65
600	60490.00	9.93	1.10
720	97361.67	9.19	1.02
840	148434.00	28.39	3.15
960	246523.00	40.87	4.54
1080	493374.83	33.94	3.77
1200	548083.67	44.50	4.94
1320	672265.83	51.96	5.77
1440	788850.33	58.07	6.45
1560	996826.00	58.34	6.48
1680	1279097.00	57.32	6.37
1800	2161382.33	42.18	4.69
1920	2055199.00	54.79	6.09
2040	2448036.83	54.19	6.02
2160	3050396.17	52.99	5.89
2280	6322217.17	30.05	3.34
2400	9563559.17	23.17	2.57
2520	15335888.67	16.83	1.87
2640	20662088.33	14.40	1.60
3000	40832635.00	9.85	1.09
3360	62151501.00	8.36	0.93
3720	87458806.00	7.28	0.81
4080	87458806.00	8.63	0.96
4440	116655204.00	7.47	0.83
4800	156450100.00	6.32	0.70
5160	199608016.00	5.54	0.62
5520	249587944.00	4.90	0.54
5880	312132568.00	4.30	0.48
6240	375312560.00	3.89	0.43

Raw statistical data for 9 core matrix multiplication

Table 6.3

16 Core Statistics			
Matrix Length	Runtime	speed up	Efficency
120	1807.60	3.01	0.19
240	22128.40	1.06	0.07
360	16439.60	8.28	0.52
480	18461.80	11.40	0.71
600	32768.60	18.32	1.15
720	57297.40	15.62	0.98
840	89176.20	47.26	2.95
960	127844.40	78.82	4.93
1080	180265.20	92.88	5.80
1200	235917.80	103.38	6.46
1320	339606.20	102.85	6.43
1440	658115.20	69.60	4.35
1560	578585.20	100.51	6.28
1680	666312.00	110.04	6.88
1800	1203345.00	75.77	4.74
1920	1048587.20	107.39	6.71
2040	4174078.20	31.78	1.99
2160	1951290.40	82.84	5.18
2280	1897853.80	100.09	6.26
2400	2827818.20	78.37	4.90
2520	2549796.20	101.21	6.33
2640	3430801.40	86.70	5.42
3120	10260708.40	43.02	2.69
3600	29714017.60	20.12	1.26
4080	74977886.40	10.06	0.63
4560	86369313.60	10.55	0.66
5040	120278652.80	8.88	0.55
5520	163741404.80	7.48	0.47
6000	207248790.40	6.66	0.42
6480	268290771.20	5.73	0.36
6960	335653248.00	5.05	0.32
7440	419832441.60	4.41	0.28

Raw statistical data for 16 core matrix multiplication

Table 6.4

25 Core Statistics			
Matrix Length	Runtime	speed up	Efficency
120	24051.17	0.23	0.01
240	11051.67	2.13	0.09
360	52583.00	2.59	0.10
480	106973.50	1.97	0.08
600	110726.17	5.42	0.22
720	199005.50	4.50	0.18
840	154338.17	27.31	1.09
960	531185.17	18.97	0.76
1080	493194.17	33.95	1.36
1200	875827.67	27.85	1.11
1320	757527.67	46.11	1.84
1440	598900.33	76.48	3.06
1560	1018206.50	57.11	2.28
1680	982701.17	74.61	2.98
1800	1514415.33	60.20	2.41
1920	1482574.83	75.95	3.04
2040	1652150.00	80.30	3.21
2160	1767076.50	91.47	3.66
2280	1699179.83	111.79	4.47
2400	2125914.83	104.25	4.17
2520	3115384.17	82.83	3.31
2640	3573438.83	83.24	3.33
3240	4402756.40	109.14	4.37
3840	14667506.20	46.10	1.84
4440	36157344.00	24.11	0.96
5040	72102244.80	14.81	0.59
5640	105465446.40	11.98	0.48
6240	146684464.00	9.95	0.40
6840	200119664.00	8.27	0.33
7440	261484960.00	7.08	0.28
8040	333422480.00	6.14	0.25

Raw statistical data for 25 core matrix multiplication