Kurt Andersen

CS 615

PA4 Matrix Multiplication

May 4, 2017

Quick Reference:
Section 1 - Sequential Report
Section 2 - Changes Made to Parallel Code
Section 3 - Parallel Report

Section 1:

Introduction:

For this project we were assigned to create a sequential program to conduct matrix multiplication. The purpose of this project is to have an algorithm that has a terrible run time complexity when run sequentially, and eventually show the massive speed-up we can get when changing it to run in parallel.

Process:

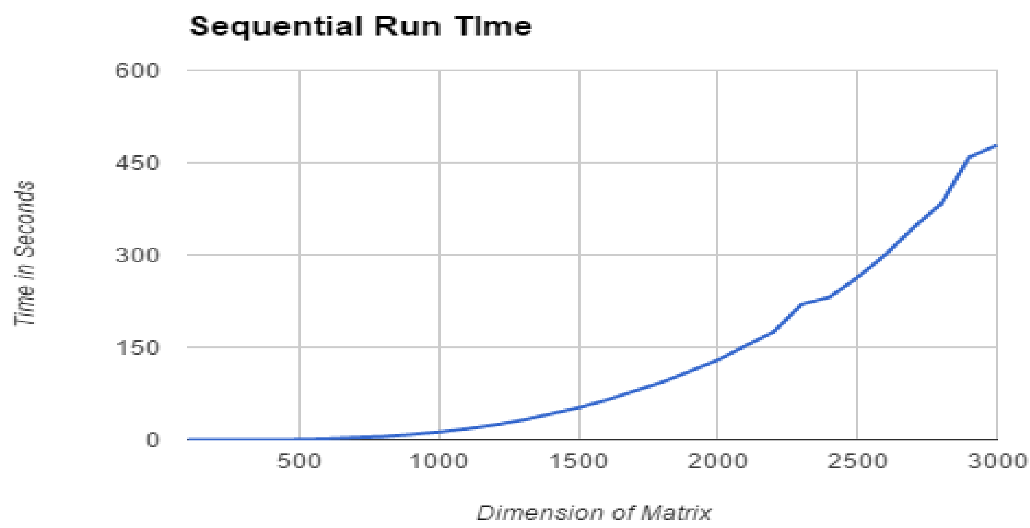
When I wrote the code for the sequential portion of this project I used google to find a simple algorithm. I then used the same idea from the online code and implemented it with the mpi library as well. I read values into my matrices in line with a constant seeded time. Then I started my timer once values were read into both matrices. From there I used a triple nested loop that found the appropriate solution matrix.

The only problem I ran into when doing the sequential portion was an error in my own code. When I was finding the sum of all the products for each cell in the matrix, I had forgot to make it find the sum of all the products. I was only taking the last multiplication for each cell. It was easily noticed and quickly fixed.

Results:

The data I gathered for the sequential portion of this project holds true to the N^3 time complexity of a standard matrix multiplication sequential algorithm. The time taken to complete each step increases exponentially. Overall this was a successful test to gather the proper data and set a good baseline for when I start getting data for the parallelized version.

Sequential Runtime Data			
Dimension	Time	Dimension	Time
100	0.002855	1600	64.1704
200	0.0205755	1700	79.0898
300	0.058846	1800	93.3977
400	0.151583	1900	111.124
500	0.360497	2000	129.475
600	1.45057	2100	152.826
700	3.32793	2200	175.332
800	4.94701	2300	219.879
900	8.1845	2400	231.449
1000	12.2958	2500	263.567
1100	17.618	2600	300.304
1200	23.9488	2700	344.359
1300	31.8389	2800	383.233
1400	41.74	2900	458.639
1500	51.9035	3000	478.483



The graph on the previous page shows the growth in time compared to the dimension of the matrix being calculated. This exponential growth in this graph strongly reflects that of an N^3 time complexity. As we increase our size, the time it takes between the calculations exponentially increase. This graph has almost no outliers and will construct a strong base for the future on this project.

Conclusion/Future Work:

Overall the sequential portion of this project was a 100% success. The data gathered shows the exponential growth perfectly, and there are no crazy outliers in the data. For the future on this project, I will use what I gathered here for a strong baseline for the parallelized version of this project.

Section 2: Updates to the file and changes as per peer feedback

During the peer review process I personally did not receive too much advice on what needed changing within my code. The main bit of feedback that I got from all of my reviewers is that I need to add functions. I added main functions that drastically reduced the amount of code in my main function, as well as reducing overall lines throughout the program. I know not having functions in during the first submission is a little grotesque to view the code, but the code was operable. I was mainly in a rush to get the code done, and did what needed to be done to complete it, and in my case it was not the most efficient structure. With the extra time to make changes based on feedback I was able to make my code a lot nicer.

Another small thing that I received was to do my calculations for my rotations differently. I did not change how I did the calculations for these because I feel as though the way I have my mesh set up, and the way my math works there is no other way. When doing column rotations I have a third case, only for the 0th column. This is because the top of the 0th column is mesh number 0. In my checks for the other rotations, it is possible that the value becomes 0. I added this extra case here so when data in the 0th column is 0, it will not wrap back to the bottom of the first column. I also did not reduce the amount of barriers that I have within my code because I want to keep my code running synchronously. I want it synchronously because I like knowing that all of the processes are at the same exact spot the entire time in running the program.

Section 3: Parallel Portion of the project

Introduction:

For this parallel portion of this project we were required to conduct Cannon's Algorithm for matrix multiplication across multiple processors. Doing the parallelized version of this project should show us how using memory in and outside of the cache can affect a parallel program.

Process:

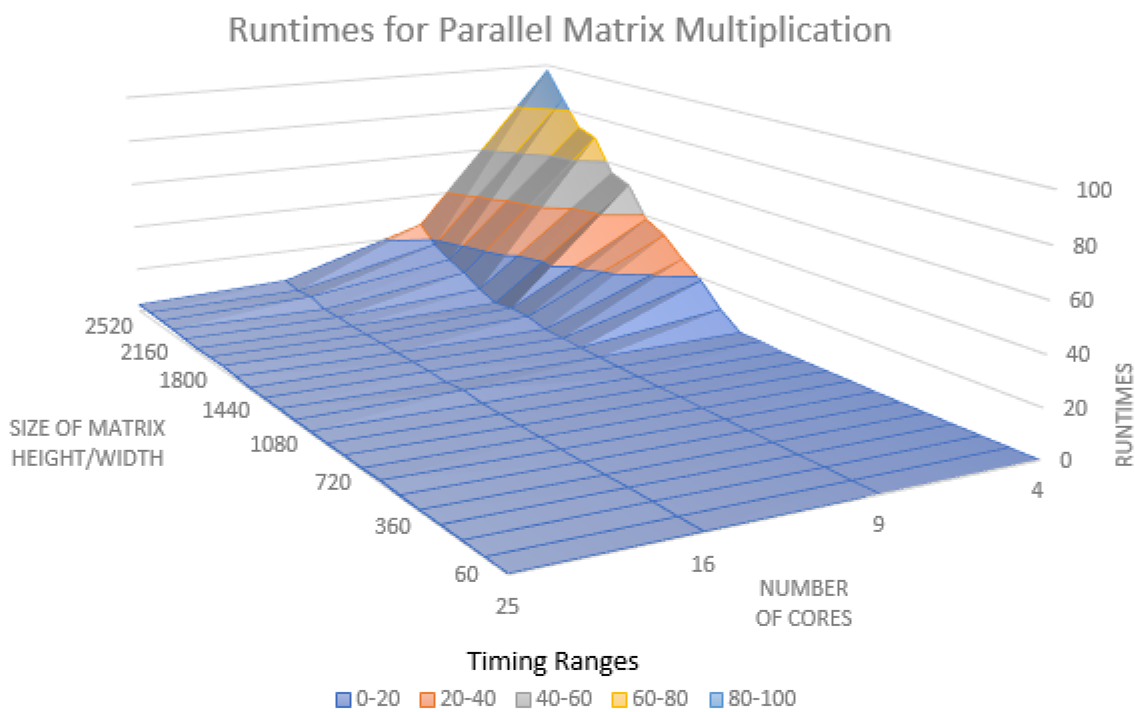
Designing the algorithm for parallel processing was a bit of a challenge for myself. I used minimal library functions for this project as well, which is probably why it was more difficult than it needed to be. When I designed my entire program, I generated values into a one dimensional array on the master processor. Once the master processor had all of the data for matrix A and B, I started to distribute the data appropriately. I treated the single dimension of processors as a mesh of processors. When distributing data, I had to do crazy calculations to properly organize the sub-matrices from the main matrix that were sent to the other processors in the mesh. Once all the data was distributed appropriately, I did Cannon's setup shifts. Once the setup shifts occurred, I did the matrix multiplication and stored the sum in each sub processors C matrix. Then I would shift the data to the left or up. After every message pass, math calculation or shift, I used a barrier to ensure my program was running synchronously on every processor. This process of multiplication then single shift occurred as many times as the square root of the total number of processors. Once this repeated process finished, then each processor sent its data back to the master process which then reorganized the data appropriately into the master solution matrix.

Points at which I ran into issues were just the math calculations I had to come up with for navigating the one dimensional arrays when I would treat them as a two or three dimensional array. Once I successfully was able to determine the appropriate calculations, it made it much simpler to conduct the rest of the project. The biggest problem I ran into though was when I first designed this program, I used standard MPI_Send and MPI_Recv, but when I was running this in parallel, any time a sub matrix had a dimension larger than 128 or so, the program would hang forever. The messages would not successfully pass. To combat this problem, I replaced each Send then Recv functions with MPI_Sendrecv_Replace. This allowed me to pass much more data than I was originally passing.

Results:

The data I gathered for this seems to match what I believed would happen when running this algorithm in parallel. In the data shown further down this page, speedup started to degrade once the processor was going outside its main cache. There were insane amounts of speed-up prior to reaching the maximum cache size, but once the process moved outside speedup started to degrade. Which at a certain point in time, speedup will eventually become almost nothing because the data set will be so massive the time reaches an infinite value.

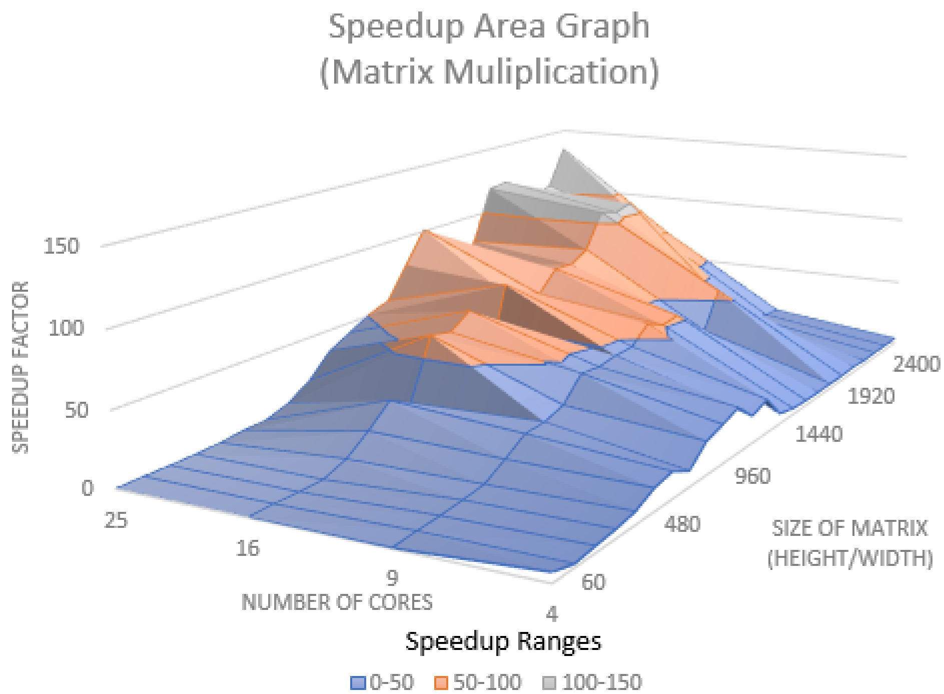
Parallel Times				
	Number of Processors			
Data Size	4	9	16	25
60	0.000447512	0.00611687	0.153469	0.0971727
180	0.00933862	0.00979114	0.0671351	0.0563738
300	0.0143974	0.0184271	0.295136	0.26704
360	0.0267501	0.0304546	0.31038	0.290245
480	0.0435627	0.0503237	0.0542867	0.299674
600	0.0980499	0.0915756	0.0798686	0.485153
720	0.189731	0.132055	0.104209	0.908463
840	0.472356	0.190133	0.207413	0.542885
960	0.550109	0.400388	0.191037	0.642828
1080	0.667149	0.384619	0.28921	0.506634
1200	0.77161	0.542227	0.339082	0.562649
1320	1.68493	0.690522	1.53129	0.626184
1440	1.77255	0.893543	0.526221	0.747456
1560	8.79318	1.1164	1.11571	0.819074
1680	18.5785	1.54351	1.2685	0.782176
1800	25.4856	1.76461	1.22247	1.35109
1920	33.6022	3.98969	1.47061	1.88427
2040	38.3043	4.23066	1.56945	1.63647
2160	51.9562	3.10514	1.63005	1.48853
2280	55.2498	6.229	2.12332	1.8411
2400	70.0454	10.7536	2.1674	2.42675
2520	72.9741	13.9755	4.63499	2.60205
2640	84.9063	18.2513	4.07214	2.84627
2760	97.4042	25.3452	5.48967	2.86357



In this graph, it shows the overall runtime of a data set on a set number of cores. The height the graph rises is how long it took the algorithm to run. While viewing the graph, you can still note a polynomial function growth. This is because there is no way to make this algorithm sub linear. The way this algorithm works, is basically this. If you are using 4 processors, then you are doing the two times half the data sets sequential time. So an example is 2400 data points on 4 processors took 70.0454 seconds. So we can look at the runtime of 1200 sequentially and multiply it by 2. The runtime of 1200 sequentially is 23.9488. The reason why it is more than double in this case is because we were going outside of the memories main cache. This cause the program to take much longer than originally suspected

Speedup

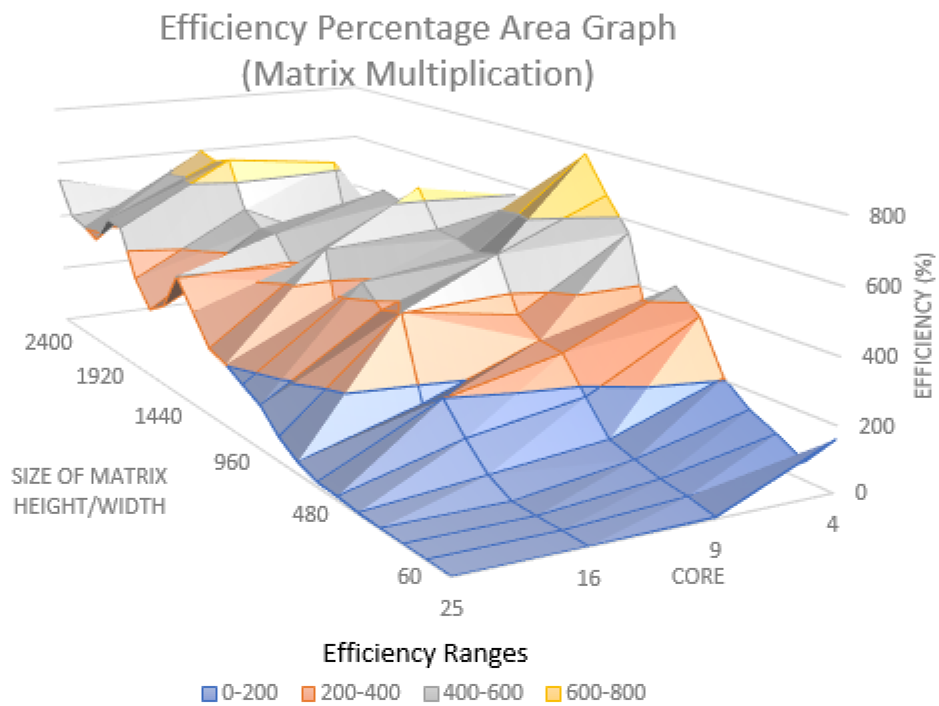
	Number of Processors			
Data Size	4	9	16	25
60	6.379717192	0.4667419775	0.01860310551	0.02938067996
180	2.203269862	2.10144069	0.3064790251	0.3649833788
300	4.087265756	3.193448779	0.1993860458	0.2203639904
360	5.666633022	4.977343324	0.4883787615	0.5222587814
480	8.275359424	7.163563093	6.640613631	1.202963887
600	14.79420173	15.84013646	18.16195601	2.989922767
720	17.54025436	25.20109045	31.93514956	3.663253209
840	10.47305422	26.01868166	23.85101223	9.112445546
960	22.35157033	30.70971158	64.36344792	19.12766712
1080	26.40789389	45.80636942	60.91767228	34.77461047
1200	31.03744119	44.16747967	70.62834359	42.56436962
1320	18.89627462	46.10845129	20.79220788	50.84591749
1440	23.54799583	46.71291701	79.32028558	55.84275195
1560	7.297746663	57.47975636	57.51530416	78.34505796
1680	4.257060581	51.2402252	62.34907371	101.1150943
1800	3.664724393	52.9282391	76.40081147	69.12766729
1920	3.307045372	27.85279057	75.56320166	58.97456309
2040	3.380168806	30.60397196	82.49705311	79.11846841
2160	3.374611692	56.46508692	107.5623447	117.7886909
2280	3.979724813	35.29924546	103.5543394	119.4280593
2400	3.30427123	21.52293186	106.7864723	95.37405996
2520	3.611788292	18.85921792	56.86463185	101.2920582
2640	3.536887133	16.45384164	73.74599105	105.5079104
2760	3.934460732	15.12053564	69.80984285	133.8304983



This graph shows the comparison of the number of cores and the size of the matrix. The vertical result is the speedup factor which is calculated by taking the sequential time and dividing it by the parallel time. The results shown in this graph follow what I imagined would happen. There are huge spikes of speedup right before the processor would use memory outside its main cache. Once it goes beyond that point, the amount of time being saved by using a parallel application starts to degrade. This is because data is getting stuck on the bus between the processor and its memory when being used.

Efficiency

	Number of Processors			
Data Size	4	9	16	25
60	159.4929298	5.186021972	0.1162694095	0.1175227199
180	55.08174655	23.349341	1.915493907	1.459933515
300	102.1816439	35.48276421	1.246162786	0.8814559617
360	141.6658255	55.30381471	3.052367259	2.089035125
480	206.8839856	79.59514547	41.50383519	4.81185555
600	369.8550432	176.0015162	113.5122251	11.95969107
720	438.506359	280.0121162	199.5946847	14.65301284
840	261.8263555	289.0964629	149.0688264	36.44978218
960	558.7892581	341.2190176	402.2715495	76.51066848
1080	660.1973472	508.9596602	380.7354517	139.0984419
1200	775.9360299	490.7497741	441.4271474	170.2574785
1320	472.4068656	512.3161254	129.9512992	203.38367
1440	588.6998956	519.0324112	495.7517849	223.3710078
1560	182.4436666	638.6639596	359.470651	313.3802318
1680	106.4265145	569.3358356	389.6817107	404.4603772
1800	91.61810983	588.0915456	477.5050717	276.5106692
1920	82.6761343	309.4754508	472.2700104	235.8982524
2040	84.50422015	340.0441329	515.6065819	316.4738736
2160	84.3652923	627.3898547	672.2646545	471.1547634
2280	99.49312034	392.2138385	647.2146214	477.7122372
2400	82.60678075	239.1436873	667.4154517	381.4962398
2520	90.2947073	209.5468657	355.4039491	405.1682327
2640	88.42217833	182.8204627	460.9124441	422.0316414
2760	98.36151829	168.0059516	436.3115178	535.3219932



This graph shows the number of cores used compared to the size of the matrix dimensions. The vertical axis shows the percentage of efficiency at the current point. This is calculated by taking the speedup factor in the same position and dividing it by the number of cores that were used for the calculation. As expected, the spikes in this graph are in the same position as the speedup factor graph. Again this is due to data still being inside the processors memory cache. Once it goes outside of it, the efficiency starts to degrade.

Conclusion/Future Work:

This project was successful in generating the expected results. This is also showed me how long data can take to communicate between the processor and memory. As we start to go outside the main cache of the processor, then the calculation time starts to drastically suffer. Some of the data was a little hard to decipher, but overall it was able to give me the results I was looking for. I ended up taking the average of the same process 5 times.

I can use the information received in this project to put it towards any future parallel algorithm I use. Using similar data like this will let me determine what kind of hardware I am using and what I can expect for time degradation for parallel applications I design on it. This will allow me to maximize the resources that I will have at my disposal.