Tyler DeFoor

PA4 - Bucketsort

4/12/2017

Dr. Harris

Sequential

Sequential Implementation

This implementation of matrix multiplication is very simple. It randomly generates numbers to fill two square matrices, then multiplies them together. The matrix multiplication portion of the code is three nested for loops. The outside goes through the rows of the result, the second goes through the columns of the result, and the final goes through the rows and columns of the left and right matrices respectively. It then has the option to print out all three matrices to be sure that everything is executing appropriately.

Sequential Analysis

Matrix multiplication with this algorithm is an $O(N^3)$ algorithm due to the nested for loops. This means that the time required to calculate the matrices grows very quickly. With 1000x1000 matrices, the execution time is 12.434 seconds. At 2700x2700 the program passes the 5 minute execution mark and continues to grow from there. The numbers that are used are too small to show the $O(N^3)$ graph, but it is consistent with what an $O(N^3)$ graph looks like in its early stages.

Sequential Graph and Table

Square Matrix Size	Execution Time	Square Matrix Size	Execution Time
1000	12.434	2000	130.444
1100	17.7073	2100	148.932
1200	24.1395	2200	176.078
1300	33.8468	2300	218.29
1400	42.2615	2400	227.498
1500	52.0867	2500	257.164
1600	65.2436	2600	296.043
1700	76.8786	2700	337.27
1800	92.5956	2800	374.922
1900	108.398	2900	432.798

Parallel

Parallel Implementation

This program implements Cannon's algorithm for parallel matrix multiplication. It requires a square matrix, a square number of processors, and for the width of the matrices to be divisible by the square root of the number of processors. This is due to the fact that the processors are formed into a grid, with each receiving its own row and column number. The matrix is then equally distributed to all of the processors.

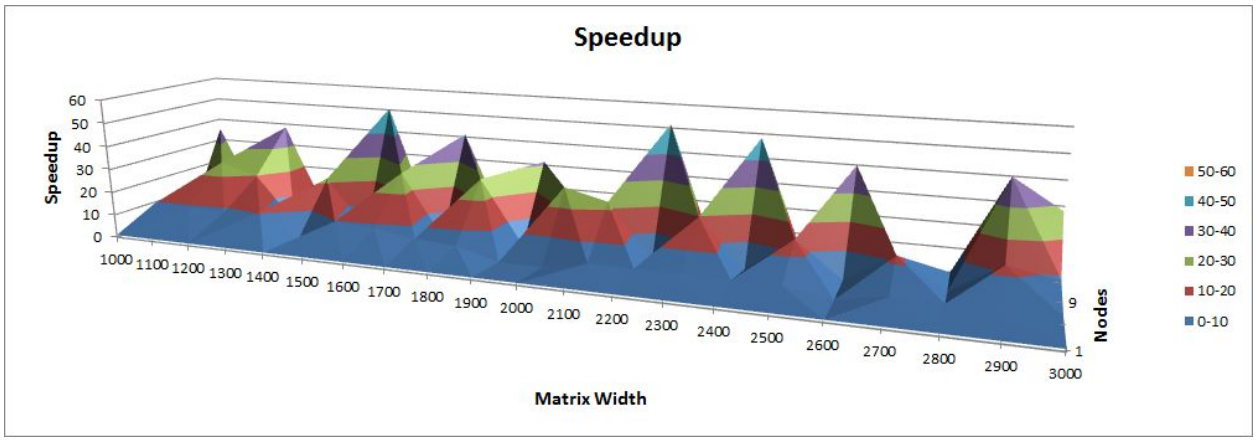
After the initial distribution, the matrices are preprocessed. All of the rows in the left matrix are shifted left by their row number, and all of columns in the right matrix are shifted up by their row number. For example, processor 2,3 will shift their left matrix's rows two to the left in the processor grid, and it will shift the right matrix's columns up three times in the processor grid. After the preprocessing, every processor's left and right matrices are multiplied together. The left matrix is passed to the processor's left neighbor, and the right matrix is passed to the processor's upper neighbor. This is repeated N - 1 times, where N is the square root of the number of processors.

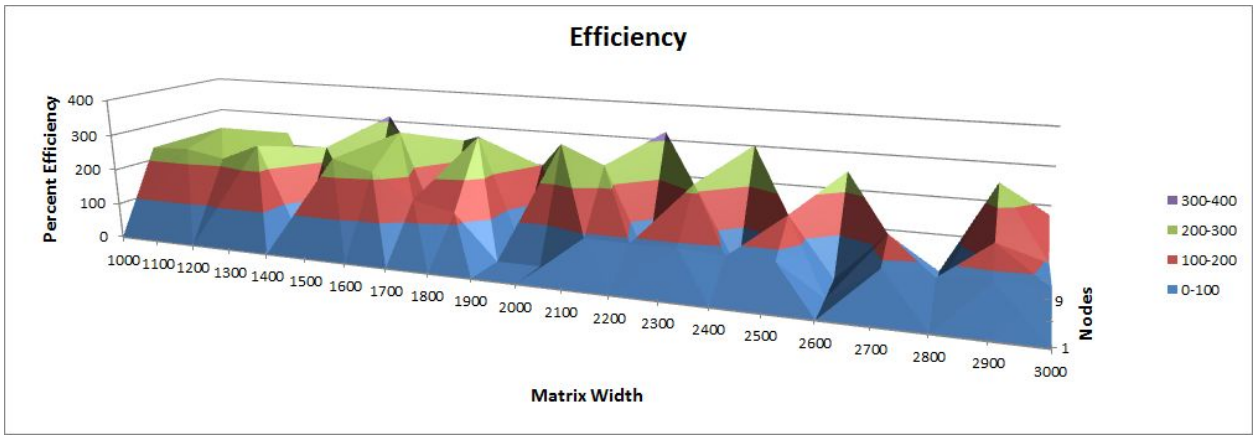
Parallel Analysis

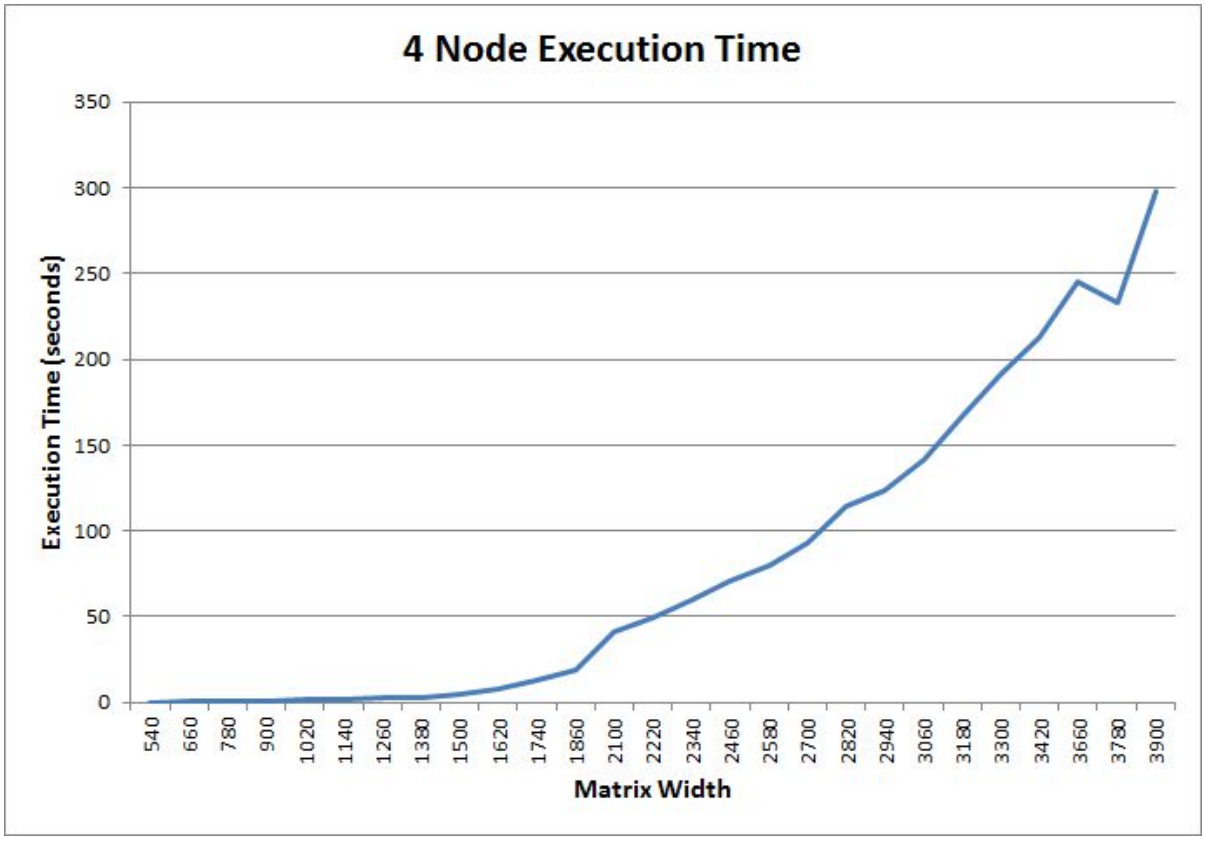
Matrix multiplication is an $O(N^3)$ algorithm, most of which is parallelizable. This means that the speedup and efficiency of adding cores is very large. As shown in the charts below, getting superlinear speedup was very common. It did not take long for the sequential portion to reach the five minute limit, so the timings are in comparison to a maximum of a 3,000 by 3,000 matrix. Many of the timings are not exactly the same, as the parallel could not work on the increments of 100 like the parallel did. The discrepancies are not large and each was compared against its closest matrix size.

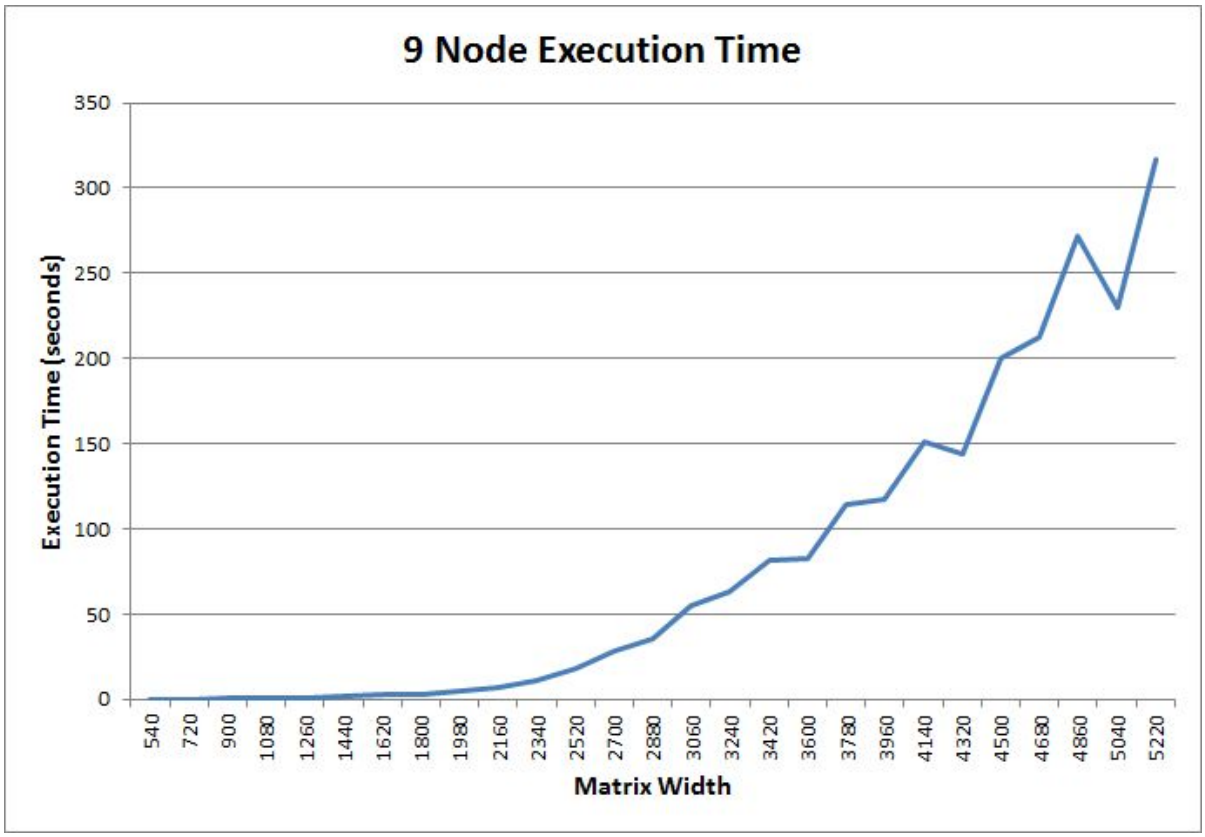
The greatest comparative speedup came from the 2,200 by 2,200 matrix on 16 cores, with a speedup of 51.6. This was also the greatest efficiency, at 322.5023%. These superlinear speedups are due to low message size and the number of processors working on each task. Having 16 processors working on the same matrix that a single processor would is a huge time save with an $O(N^3)$ algorithm.

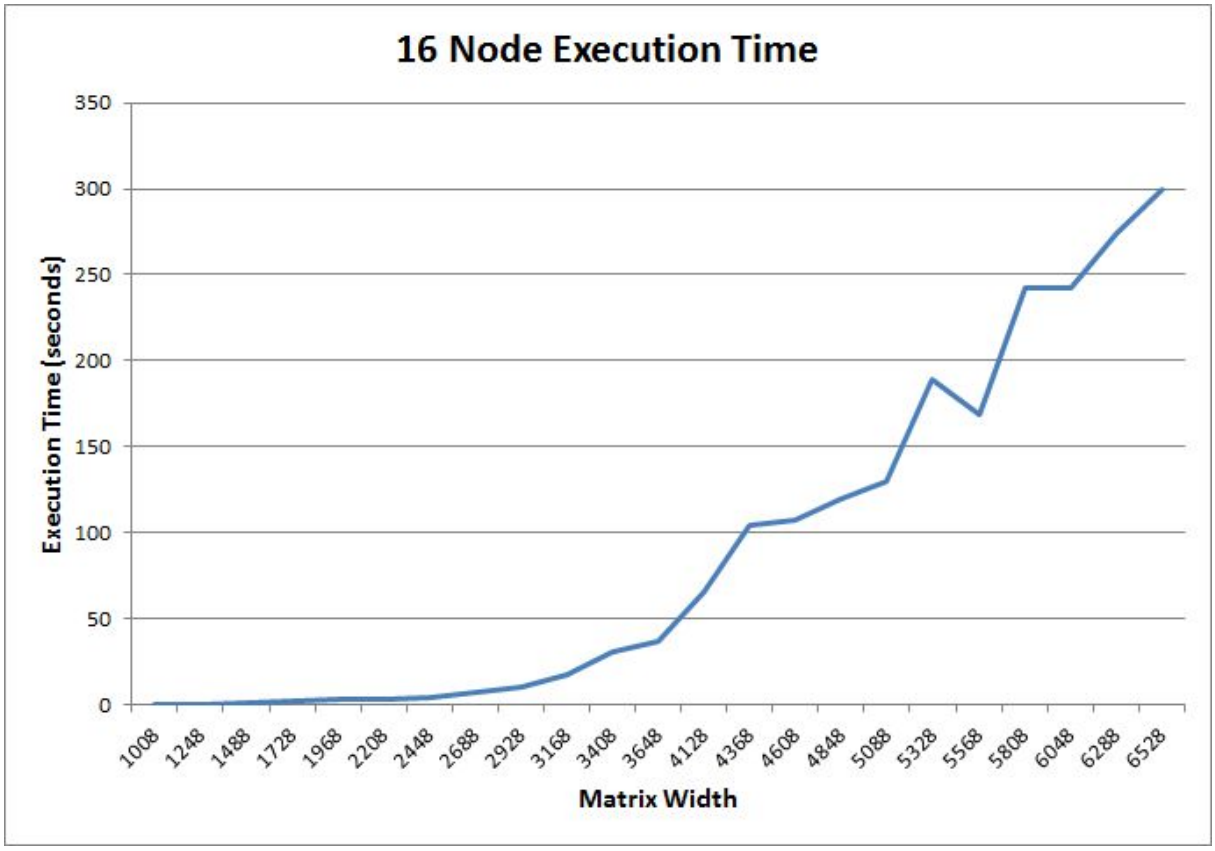
Parallel Graphs






4 Node Execution Time
Execution Time (seconds)
0
50
100
150
200
250
300
350
540
660
780
900
1020
1140
1260
1380
1500
1620
1740
1860
2100
2220
2340
2460
2580
2700
2820
2940
3060
3180
3300
3420
3660
3780
3900
Matrix Width


9 Node Execution Time
Execution Time (seconds)
0
50
100
150
200
250
300
350
540
720
900
1080
1260
1440
1620
1800
1980
2160
2340
2520
2700
2880
3060
3240
3420
3600
3780
3960
4140
4320
4500
4680
4860
5040
5220
Matrix Width


16 Node Execution Time
Execution Time (seconds)
350
300
250
200
150
100
50
0
1008
1248
1488
1728
1968
2208
2448
2688
2928
3168
3408
3648
4128
4368
4608
4848
5088
5328
5568
5808
6048
6288
6528
Matrix Width

Future Works

I would like to revisit this in the future for two reasons. The first is to improve the program to work on matrices that are indivisible by the square processor size. The second is that the algorithm is already very scalable, so using it on clusters with more processors would be interesting to see.