Adam Montano

CS 415 – Spring 2017

PA4 – Matrix Multiplication: Parallel

Introduction

Matrix multiplication is a remedial process that can be parallelized by a perfect square number of processors. In this report, we will analyze the effect parallelizing has on matrix multiplication in comparison to the sequential implementation. The parallel implementation uses Cannon's algorithm to multiply each matrix and simulates a torus mesh of nodes. In other words, each processor directly links to each other as if they were aligned as a matrix.

Parallel Implementation

The parallel implementation uses Cannon's algorithm and a simulation of a torus mesh of nodes. The two square matrices are read in and sub matrices (of size square root of the number of processors) are allocated to each processor. Initially, each row and column are swapped a certain amount of times. Then, each sub matrix that is allocated to each processor sends left and up one at a time and finally is multiplied to a resultant matrix. After a certain amount of repetitions, the final matrix is the result.

Below is table of run times with a corresponding matrix size.

Matrix Size (Height x Width)	**Run Time (Seconds)**				
	Sequential	**4 Processors**	**9 Processors**	**16 Processors**	**25 Processors**
240	0.02941	0.007013	0.02609	0.008204	0.01159
480	0.329189	0.044889	0.043265	0.028501	0.136448
720	3.04832	0.192098	0.118407	0.07215	0.279387
960	10.8757	0.562417	0.396117	0.143057	0.155957
1200	25.0285	0.971098	0.464044	0.411	0.43637
1440	44.2823	1.93064	0.984008	0.489555	0.452278
1680	72.4629	8.83625	1.31387	1.12863	0.770952
1920	110.112	20.6883	3.80349	1.31868	1.42999
2160	159.86	35.2613	3.14766	1.65138	1.37366
2400	222.772	49.4378	10.2936	2.32633	3.41771
2640	315.989	68.4373	18.5459	4.83457	3.13012
2880	416.056	89.7001	32.4814	4.97431	4.81832
3120	531.605	115.79	45.3453	13.6524	5.35135
3360	724.328	146.013	60.2411	18.8941	9.09178
3600	969.022	186.678	76.2681	33.4103	10.6148

Below are graphs comparing the run times of each of these as well as a graph for speed up and efficiency.

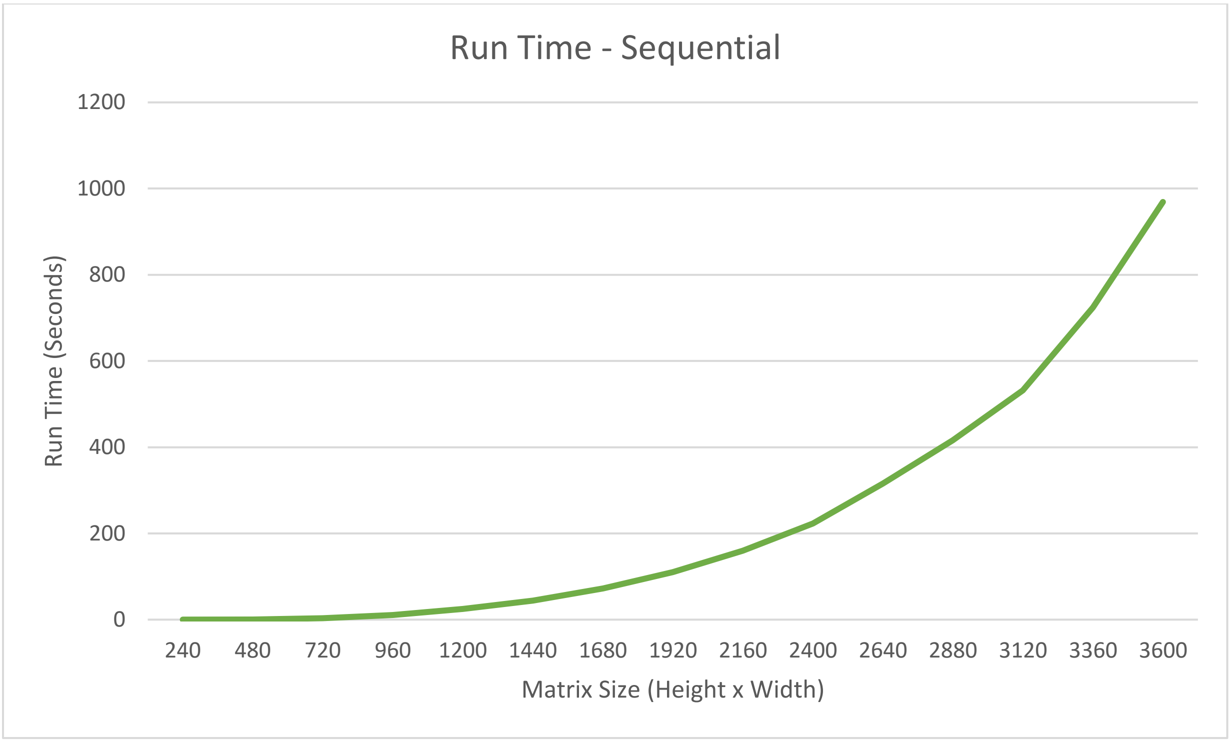

Figure 1 - Sequential Run Time

It is evident by the surface plot (Fig. 1) that the time complexity of the sequential matrix multiplication strategy is polynomial. Now let’s compare it to varying processors using Cannon’s algorithm.

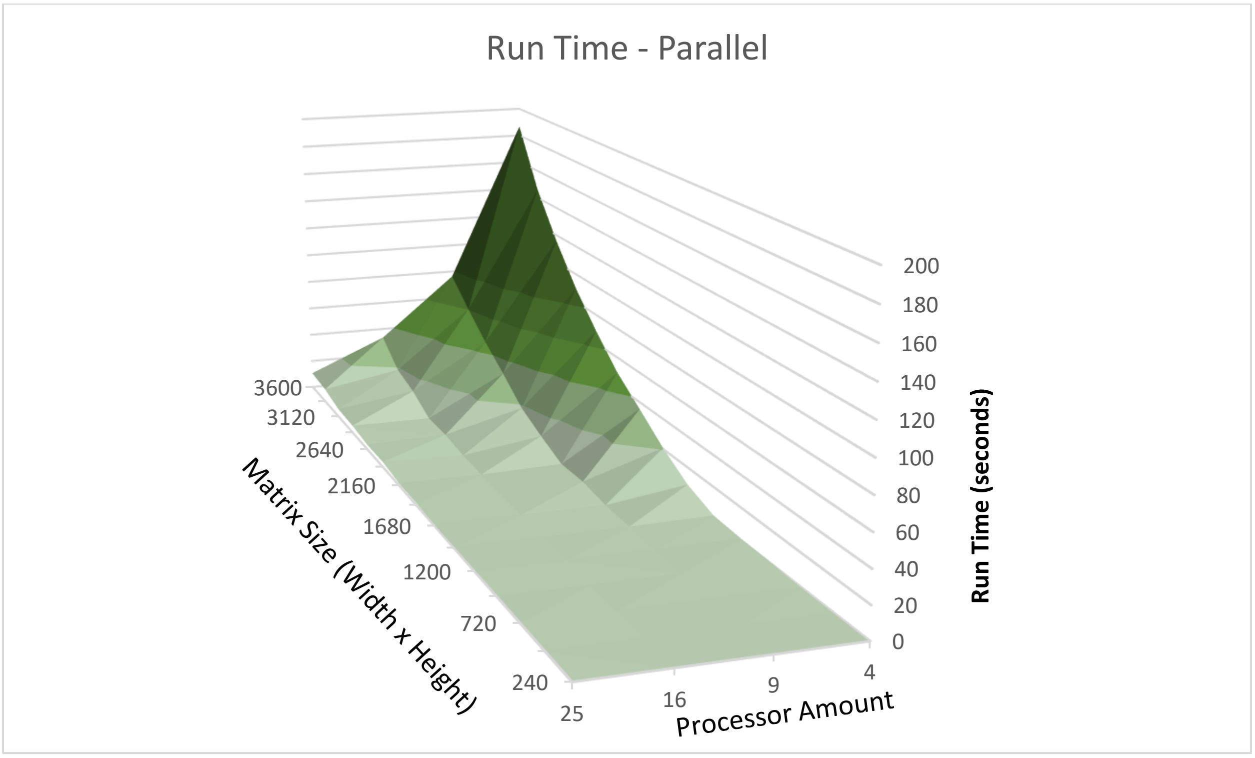

Figure 2 - Parallel Run Time

The above surface graph shows the run times of varying processors and matrix sizes. It is visually evident that a large number of processors handles the work load more effectively than a smaller amount. The drop off in time is in plain sight. Even at the largest matrix size of 3600 x 3600, twenty-five cores complete the matrix multiplication nearly 10 times faster than four cores.

Below is a graph for speed up.

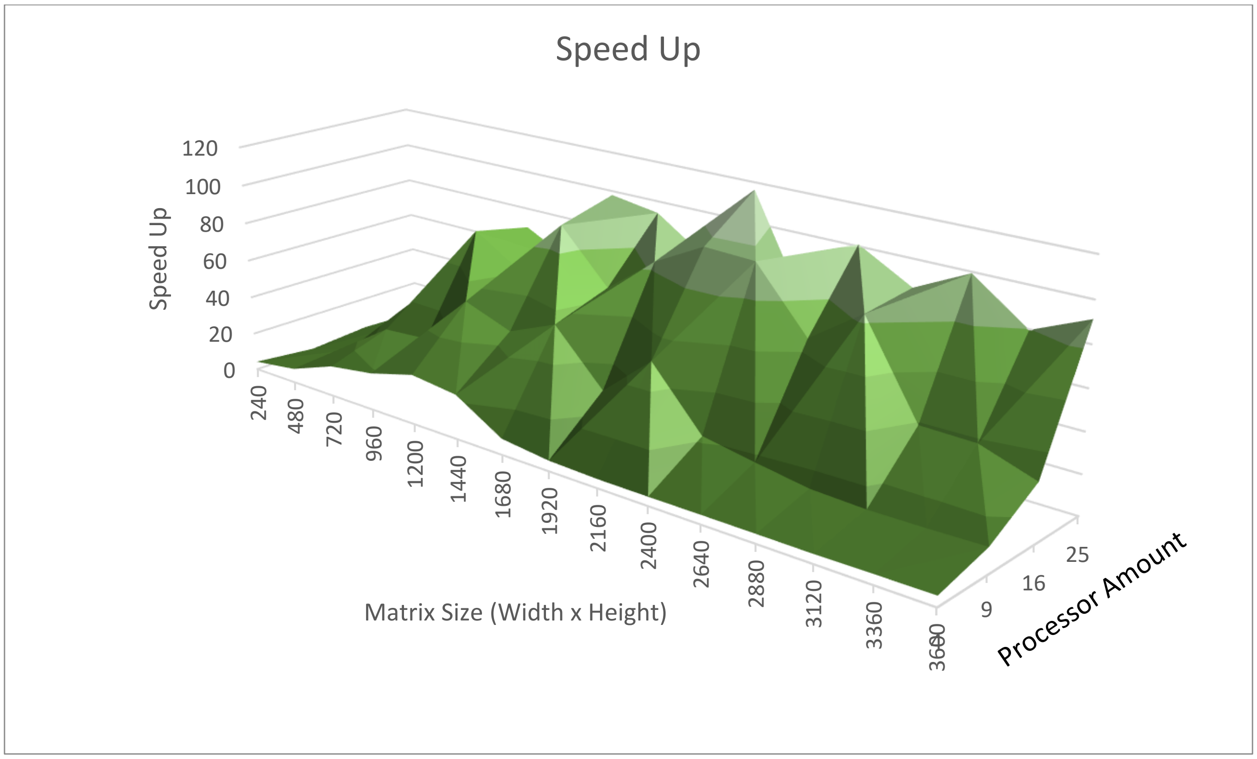

Figure 3 - Speed Up

Clearly, we are experiencing a large amount of speedup, superlinear speedup even. The general direction of the graph shows that speed up is directly correlated with the number of cores. Superlinear speedup is explained through either the software or hardware. It is possible that the sequential implementation uses a suboptimal algorithm, which is true to some extent. There exist better matrix multiplication algorithms (such as Strassen's). Another possible explanation is the nature of the architecture of the system. The caching effect can provide a possible explanation to the superlinearity of the implementation. Each node and "box" has a cache and the greater the number of nodes and boxes implies a greater number of caches. As there are more caches to work with, memory accessing becomes rudimentary and is accessed very quickly, at least much faster than the sequential implementation. Another factor is that the greater the number of processors, the smaller the size of the sub matrix each node must work with. Any combination of these factors can provide an answer but it is certain that the transformative law of quantity into quality is confirmed.

Below is a graph of efficiency.

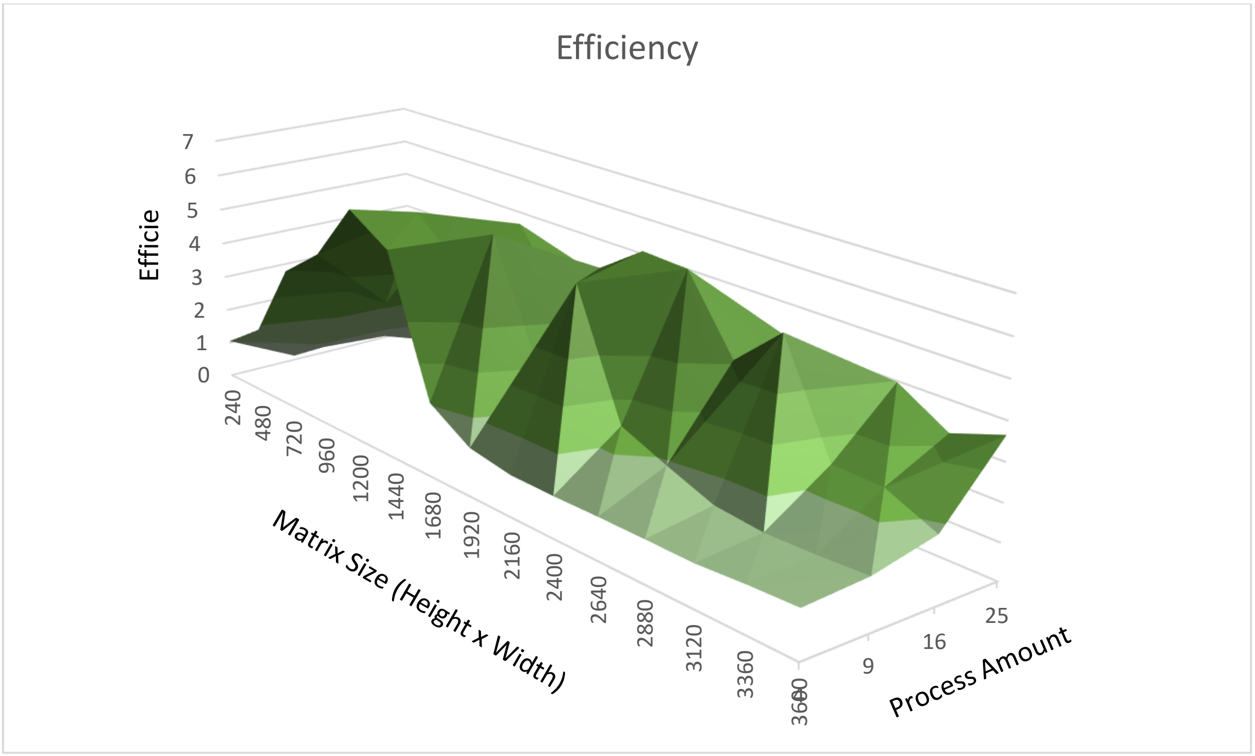

Figure 4 - Efficiency

Clearly, the efficiency fluctuates greatly in this graph. However, throughout most the graph we observe that there is almost always an extremely high efficiency. No doubt that this is due to the excessive amounts of speed up explained previously. Overall, the efficiency is relatively low during the smaller matrix sizes, increases during mid-sized matrices and drops off later. Especially for the implementation with four processors, the efficiency decreases with larger matrix sizes. It is possible that the smaller processor amounts cannot use the cache as efficiently as the others and as a result require a longer time to access memory. The "goldilocks" zone for cache size must be around 1000 x 1000 integers.

Issues

The most glaring issue was handling the deadlock that occurred when trying to rotate the sub matrices up or left. When one processor needed to send its matrix to the processor adjacent, the adjacent processor was doing the same for its adjacent processor in the mesh. Since all the processors were waiting on each other, we reached a deadlock. The solution was to use a non-blocking send. This way the processors "juggled" the sub matrices efficiently. Each processor instantly sent their sub matrix to the ether and waited for a matrix as soon as possible.

Conclusion

The parallel implementation greatly outperforms the sequential implementation. The result is superlinear speed up and an excessive efficiency. Using Cannon's algorithm is much better than a sequential, iterative matrix multiplication. The superlinear speedup can be explained by a number of hardware or software factors. Deadlocks were avoided by using a "juggle" method of exchange submatrices.