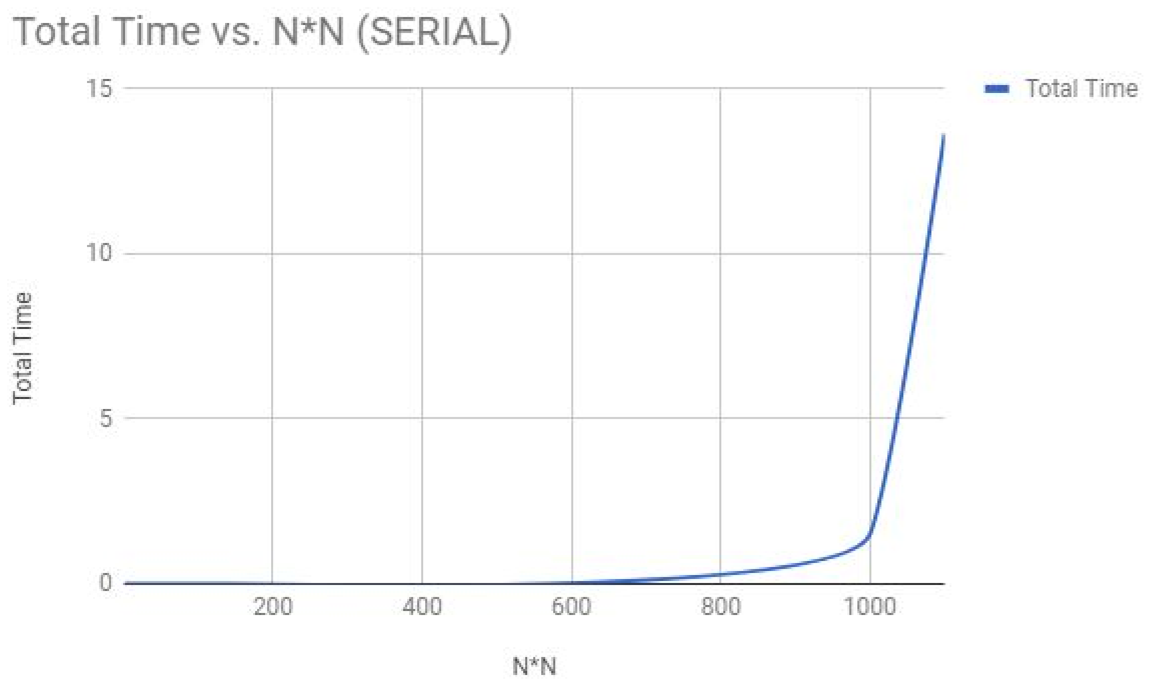
Code sources:
Serial Matrix Multiply: http://sandbox.mc.edu/~bennet/parallel/code/smmult_c.html
OpenMP Matrix Multiply: https://computing.llnl.gov/tutorials/openMP/samples/C/omp_mm.c
MPI Cannon Matrix Multiply: https://github.com/cstroe/PP-MM-A03/tree/master/code
MPI Summa Matrix Multiply: http://kayaogz.github.io/teaching/ (Look for tp4-code.tgz and solution)

Serial Matrix Multiply:
(To test, pass in two values to get size*size)

N*N	Total Time
1	0.000000
10	0.000001
100	0.001048
1,000	1.503135
1,100	13.641109



A serial matrix multiply performs best for very small array sizes. It runs very fast for sizes ranging from 1x1 to 10x10. We only test until 1,100 size as the running time becomes very very long.

OpenMP Matrix Multiply:
(To test, pass in 3 values all equal to each other: 1000 1000 1000 for N*N = 1000)

1 thread:

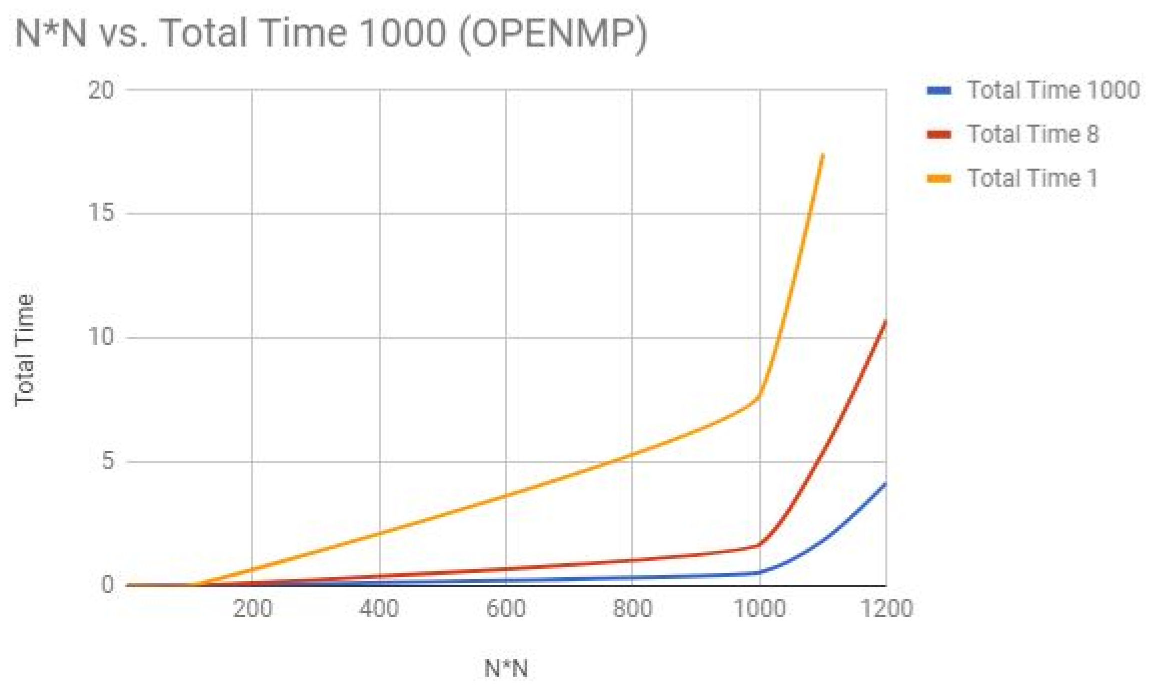
N*N	Total Time
1	0.000006
10	0.000022
100	0.001316
1,000	7.705094
1,100	17.441623

8 threads:

N*N	Total Time
1	0.000017
10	0.000167
100	0.001289
1,000	1.668558
1,100	5.409879
1,200	10.753936

1000 threads:

N*N	Total Time
1	0.002865
10	0.002788
100	0.003307
1,000	0.539878
1,100	1.831835
1,200	4.160250



OpenMP is very efficient when handling bigger array sizes. We notice that its much faster than serial in the case of 1,100 and even beats serial size 1,100 when handling size 1,200. We also see that for bigger sizes, the more threads we throw at the problem, the better.

MPI Cannon Matrix Multiply:
(To test, pass in 4 value: the size of each matrix and its power: 1000 1 1000 1 for N*N = 1000)
1 core:

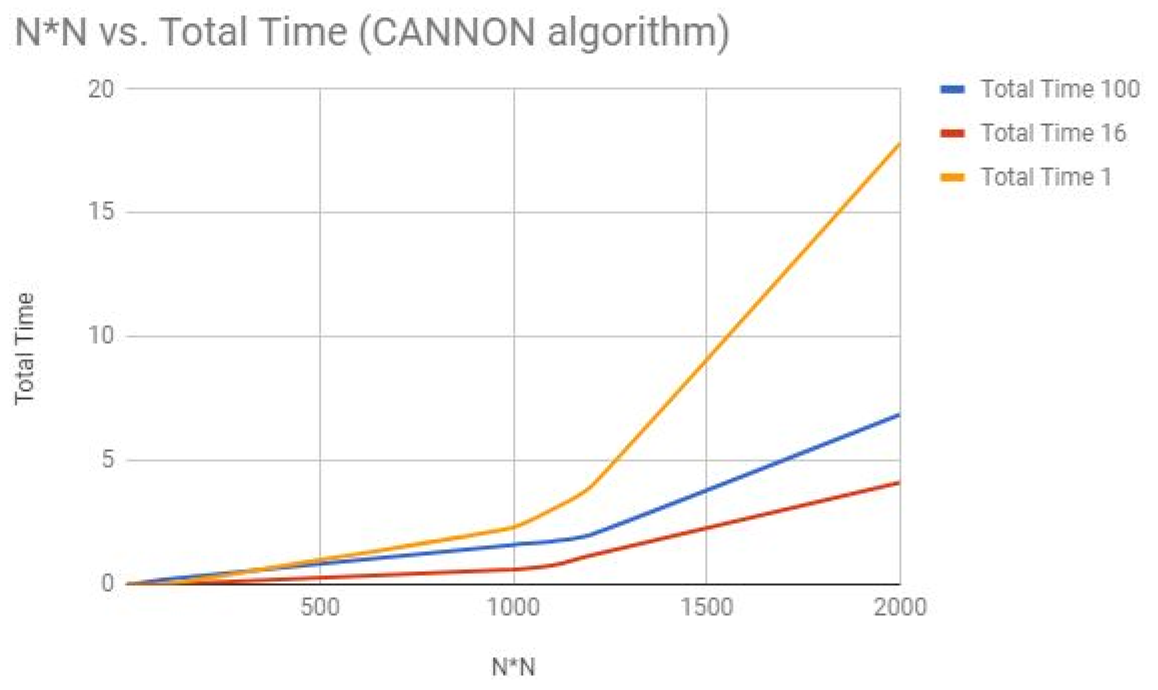
N*N	Total Time
1	0.000139
10	0.000280
100	0.0002925
1,000	2.304801
1,100	3.021477
1,200	3.938414
2,000	17.827338

16 cores:

N*N	Total Time
1	N/A
10	N/A
100	0.004739
1,000	0.593780
1,100	0.764528
1,200	1.166887
2,000	4.115526

100 cores:

N*N	Total Time
1	N/A
10	N/A
100	0.193914
1,000	1.596018
1,100	1.740383
1,200	2.002803
2,000	6.862510



MPI cannon matrix multiply is even more powerful for large arrays compared to OpenMP. We notice that the more cores we throw at the problem, the faster it completes. This is apparent at the 2000 array size as using 8 cores is much faster for this case than using a single core. We also notice that for a large number of cores, the inter-process communication overhead negates the performance improvement when testing for medium-sized arrays. We need to test for very large arrays to see an improvement but this takes a very long time.

MPI Summa Matrix Multiply:
(To test, pass in 1 value equal to N*N)
1 core:

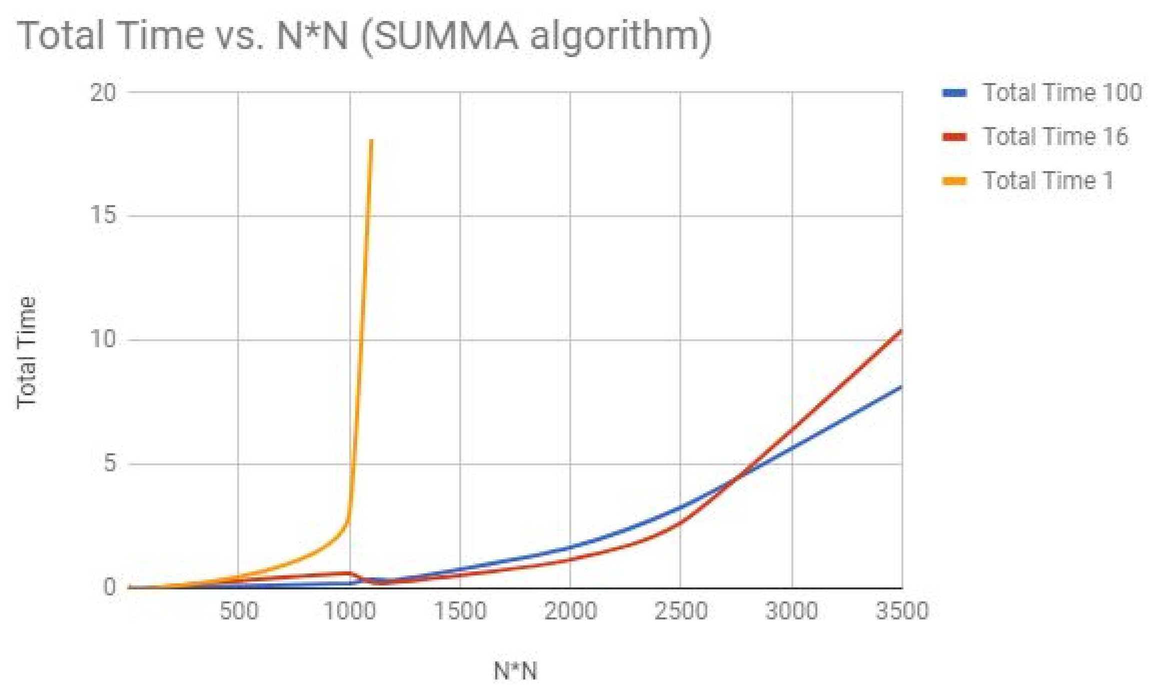
N*N	Total Time
1	0.000100
10	0.000100
100	0.001800
1,000	3.0165
1,100	18.1438

16 cores:

N*N	Total Time
1	0.000500
10	N/A
100	0.004739
1,000	0.593780
1,100	0.233500
1,200	0.236200
2,000	1.157400
2,500	2.650600
3,500	10.435900

100 cores:

N*N	Total Time
1	0.0129
10	0.00087
100	0.009400
1,000	0.194400
1,100	0.364600
1,200	0.322800
2,000	1.645900
2,500	3.270900
3,500	8.148800



MPI Summa matrix multiply is the superior matrix multiplication method. Not only does it use less resources compared to cannon, it performs better. The performance scaling relative to the increase in the number of cores is similar to the Cannon MPI algorithm but the benefits are more apparent here. Summa algorithm can handle bigger arrays in a more reasonable amount of time.

Conclusion:
The serial algorithm is best for very small arrays with a size up to 10*10. For bigger arrays, it is better to use OpenMP multithreading if we only have access to a single machine with few cores. If we do have access to a large number of cores, the SUMMA MPI algorithm is the superior algorithm for matrix multiplication. It provides visible performance benefits and uses less resources compared to Cannon.