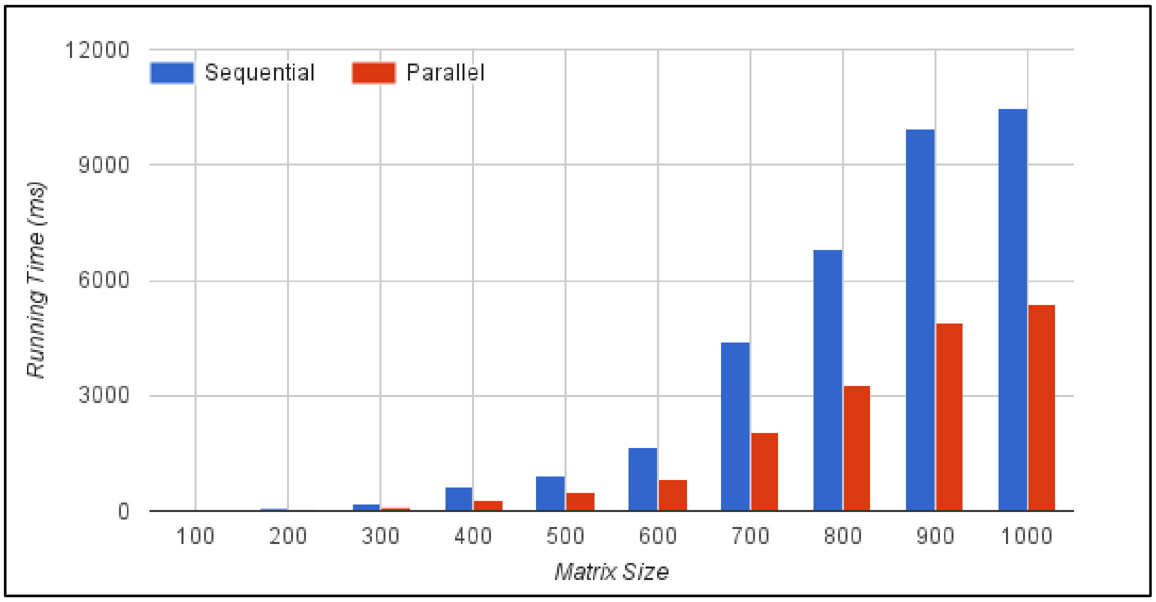
Concurrent Programming Project

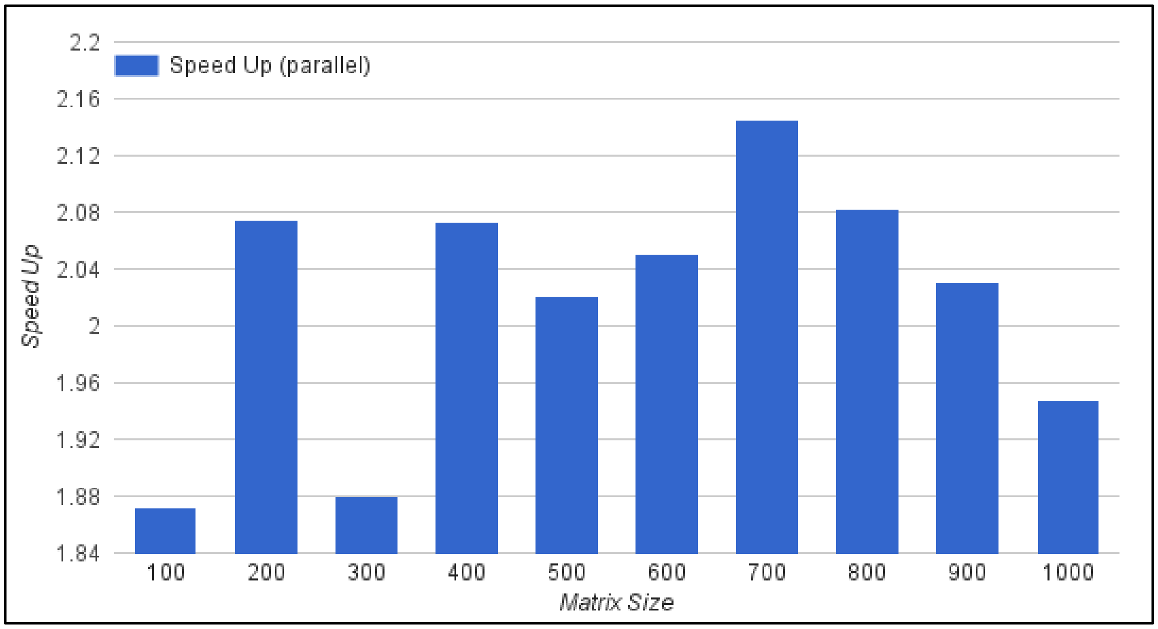
Use "parallel for" and optimize matrix-matrix multiplication to gain better performance

- Used library for C++ and Linux environment that supports "parallel for" loops.
  OpenMP

- Results of both sequential and parallel-for version programs (attached separately) while changing the matrix size *n* from 100 to 1,000 in steps of 100 shows following diagrams.
  For calculating the number of samples to be observed, initially code was run 30 times and calculated the mean and standard deviation. When calculating the number of samples with mean, standard deviation and the given accuracy, I got number of samples less than 30. So I decided to use the initial mean values (30 iterations) rather than using the mean of less number of samples.



Matrix multiplication time against increasing n.



Speed up against increasing n.

- Architecture of the CPU that I used for the evaluation as follows

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	2
Core(s) per socket:	2
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	37
Stepping:	5
CPU MHz:	933.000
BogoMIPS:	5066.85
Virtualisation:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	3072K
NUMA node0 CPU(s):	0-3

The CPU that was used for the evaluation has 2 physical cores with 2 threads per core. That means the CPU can process 4 threads simultaneously. The library (OpenMP) that was used for the parallel program by default uses number of physical threads for the parallel for function. When parallelizing the serial code, scheduler is responsible for spawning maximum 4 number of threads. As long as I don't feed the number of threads into the OpenMP, maximum number of threads will be 4(depends on the CPU architecture). When creating more threads, running time might be large because of the thread scheduling time and context switchings. So gained speed is limited to 4 threads. Average speed up for the parallel version is 2.01, according to the Amdahl's Law (n=4) parallel fraction can be calculated as 0.67. Furthermore graphs shows the reduction of running times with the matrix size. In each case, running time is nearly reduced by factor of 2. As well as maximum speedup was gained when matrix size is 700. Also graph shows equally speedup for matrix sizes 200, 400, and 800.

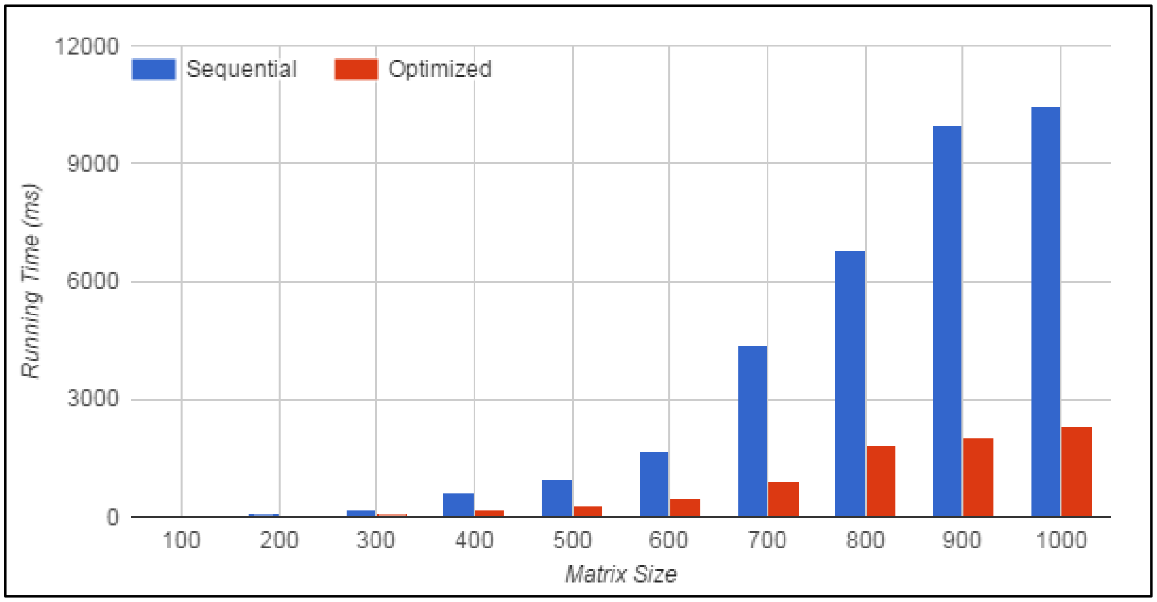
- Another two ways that can be used to optimize matrix multiplication other than parallel-for as follows.
  - **a. Cache Optimization**

    When matrix is getting larger and larger, it is hard to load entire matrix into cache at once. When considering two matrices A and B, matrix multiplication is performing with a row of matrix A and a column of matrix B. For each of element of the resultant matrix this calculation is iteratively happening. When calculating the 1000x1000 or larger matrices, it is impossible to load entire matrix into cache. For matrix B (matrix which consider column) there are several cache misses when performing calculation. Here I have to consider the spatial locality. If all elements of matrix B which are going to be accessed in near future can be loaded into cache, then I can reduce the rate of cache misses. As a solution for this issue, I transposed the matrix B and did the calculations. Then the cache misses related to matrix B are reduced and running time also can be reduced.

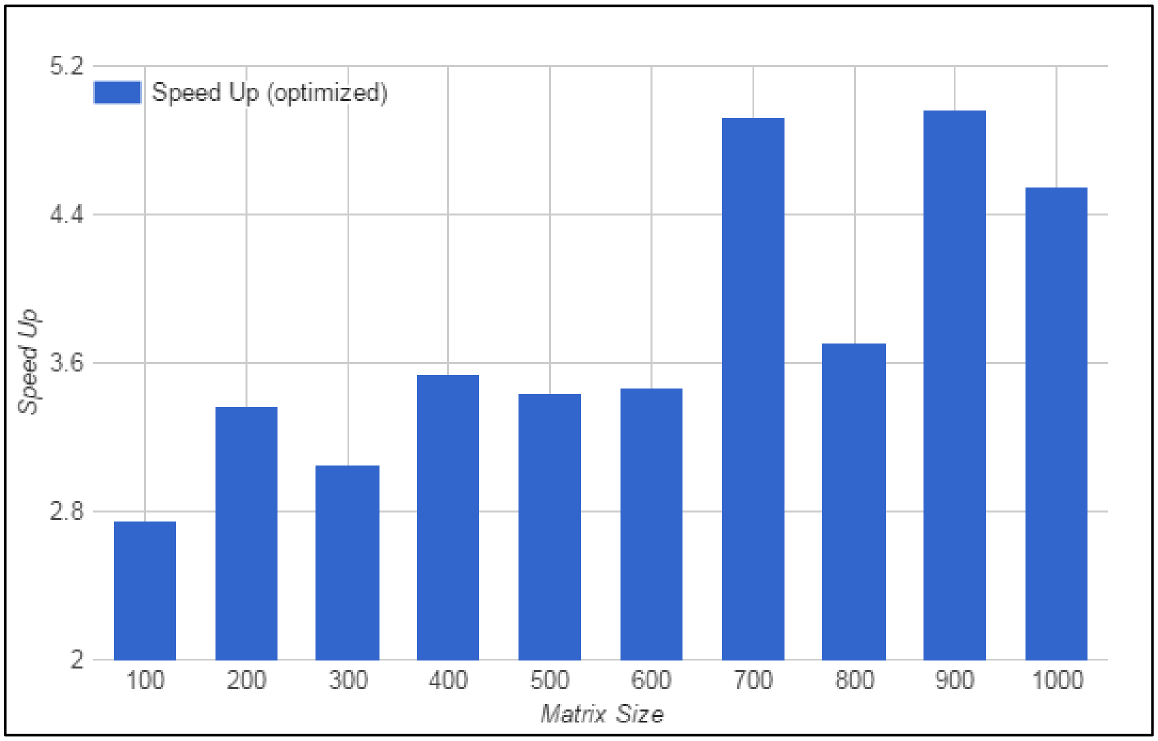
  - **b. Multithreading Strassen's Algorithm**

    The Strassen's method of matrix multiplication is a typical divide and conquer algorithm. In this case, divide the input matrices A and B and output matrix C into $n/2 \times n/2$ sub matrices. Then create 10 matrices S1, S2 ...S10, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices which was created in previous step. These 10 matrices are computed using parallel for loops with Θ (n^2) work and Θ (lgn) span. Using these sub-matrices created in above steps, I can recursively spawn the computation of seven $n/2 \times n/2$ matrix products P1, P2 …P7. For the final result, I need to compute the desired sub matrices C11, C12, C21, and C22 of the result matrix C by adding and subtracting various combinations of the Pi matrices, once again using doubly nested parallel for loops.

- **"Cache Optimization"** method is the one I was implemented in this project. Results of optimized method vs. sequential method as follows



Matrix multiplication time against increasing n.



Speed up against increasing n.

According to the observations of graphs, running time of optimized algorithm has been nearly reduced by factor of 4 with the increase of matrix size. Since this is a cache optimization, the size of the cache is directly affected to the gained speedup of the program. For high speed up values, I can assume that the system was able to load more potion from the matrix into cache at once and then the rate of cache misses were significantly reduced. According to the architecture and Amdahl's Law 4 times speedup can be gained if the program is 100% parallelized. According to the speedup obtained by optimized program, average speedup can be calculated as 3.78 which is closer to the maximum speedup that can be obtained from our CPU architecture.