Concurrent Programming Lab 4

140161F (P.R.L. Fernando)
140097M(D.G.V.M. Dayasiri)

4) Run both sequential and parallel-for versions of your program while changing the matrix size n from 200 to 2000 in steps of 200.

- Record the time to perform matrix-matrix multiplication for each execution.

Initially we used 20 values and then calculated mean and sample deviation respectively. Then we calculated needed sample size by taking the accuracy level 5% and confidence level of 95%. We used following equation to calculate the sample size.

z = 1.960
s = sample sd,
r = required accuracy
m = sample mean
n = required sample size

$$n = \left(\frac{100 * z * s}{r * m} \right)^2$$

Matrix Dimension	Serial (s)	Parallel (s)
200	0.043323	0.0356243
400	0.345437	0.236883
600	1.25438	0.898377
800	3.00053	2.2899
1000	6.71874	4.5796
1200	13.6369	8.25647
1400	21.6993	14.0824
1600	33.9808	22.6629
1800	51.1312	33.5222
2000	73.0928	48.0586

Table 1: Average time for serial and parallel execution.

- Collect sufficient number of samples for each configuration of n, while making sure your performance results are within an accuracy of ±5% and 95% confidence level.

- For serial Implementation

Dimension	Used Sample size	Sample mean	Sample standard deviation	Required sample size
200	20	0.040426	0.00208416	5
400	20	0.346936	0.00634182	1
600	20	1.27073	0.012413	1
800	20	3.03956	0.0986063	2
1000	20	6.79399	0.0562541	1
1200	20	13.3873	0.082815	1
1400	20	21.5861	0.135336	1
1600	20	34.5948	0.206066	1
1800	15	56.0104	0.45328	1
2000	15	80.9087	0.745822	1

Table 2: Serial execution time with required sample size

- For parallel implementation

Dimension	Used Sample size	Sample mean	Sample standard deviation	Required sample size
200	20	0.0282433	0.0010858	3
400	20	0.234198	0.00142329	1
600	20	0.925381	0.027037	2
800	20	2.25853	0.0338886	1
1000	20	4.59211	0.07734	1
1200	20	8.3394	0.0517098	1
1400	20	14.1211	0.044551	1
1600	20	22.1996	0.0384402	1
1800	15	33.7306	0.184987	1
2000	15	48.0129	0.0861737	1

Table 3: Parallel execution time with required sample size

- Plot the matrix-matrix multiplication time against increasing n.
- Average time calculation

Dimension	Average time for Serial	Average time for parallel
200	0.040426	0.0282433
400	0.346936	0.234198
600	1.27073	0.925381
800	3.03956	2.25853
1000	6.79399	4.59211
1200	13.3873	8.3394
1400	21.5861	14.1211
1600	34.5948	22.1996
1800	56.0104	33.7306
2000	80.9087	48.0129

Table 4: Serial time vs parallel time

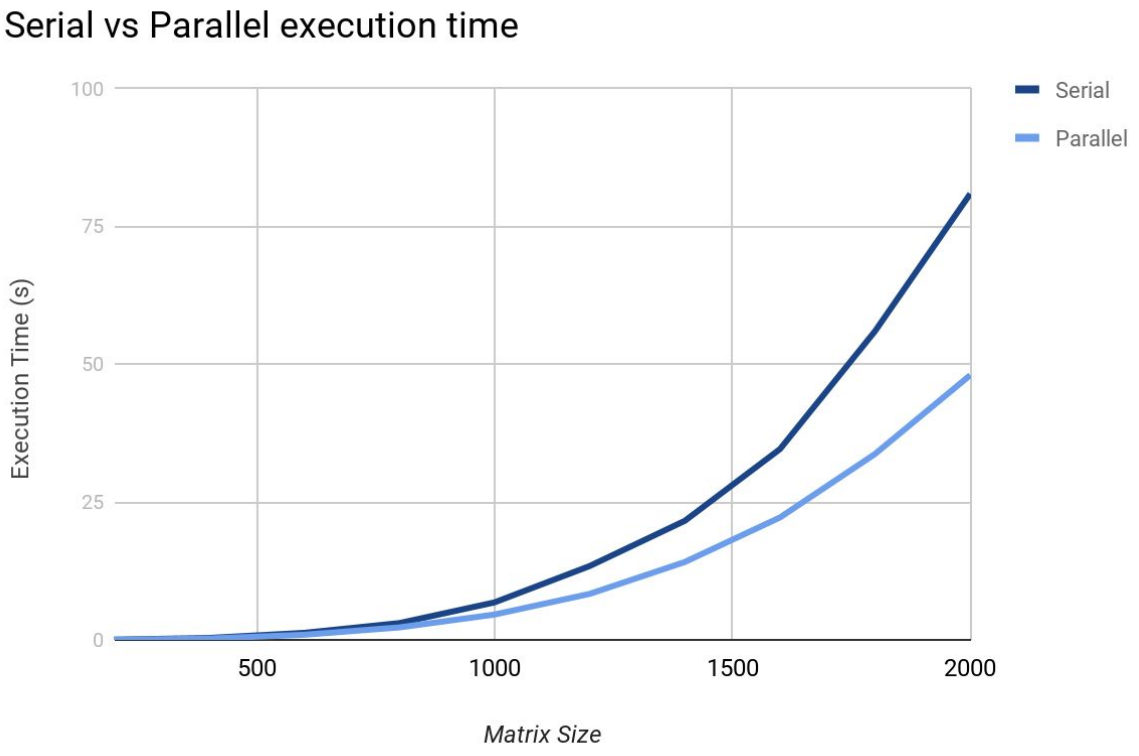
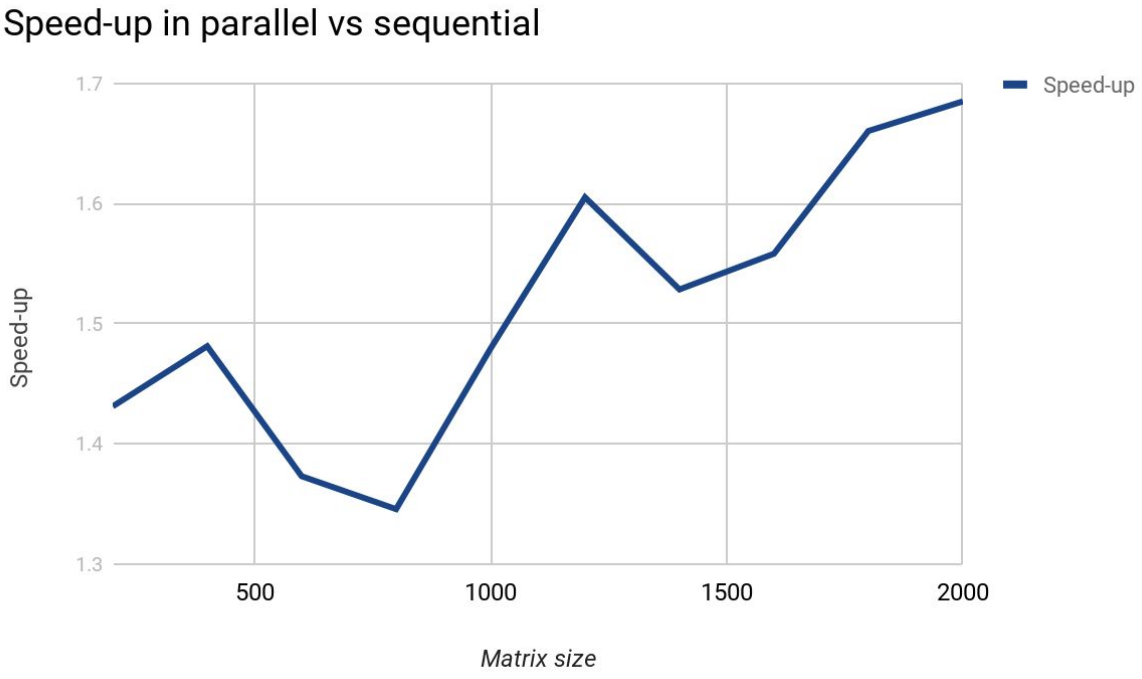


Figure 1: Graphs of Serial Parallel execution time

- Plot speed up against increasing n.
- Speed up calculation table

Dimension	Avg time for serial	Avg time for parallel	Speed up
200	0.040426	0.0282433	1.43134832
400	0.346936	0.234198	1.481379004
600	1.27073	0.925381	1.373196554
800	3.03956	2.25853	1.345813427
1000	6.79399	4.59211	1.479491998
1200	13.3873	8.3394	1.605307336
1400	21.5861	14.1211	1.528641536
1600	34.5948	22.1996	1.558352403
1800	56.0104	33.7306	1.6605219
2000	80.9087	48.0129	1.685145034

Table 5: Speed-up calculation between serial and parallel execution time



Graph 2: Speed-up for parallel for number of matrix size.

5) Find out about the architecture of the CPU that you used for the evaluation.

- Find out CPU model, no of cores, no of hardware-level threads, cache hierarchy, hardware optimizations, etc.

  CPU specification

CPU	4x Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz
# Cores	2
# Hardware Threads	4
Cache	L1 128KiB L2 512KiB L3 4MiB

- Can you justify the gained speed up knowing the architecture of the CPU?

According to the figure 2 we can see that the gap between serial and parallel implementations are gradually increasing. For small amount of values both serial and parallel implementations take the same amount of time because thread creation will take some time and the speed up gained by execution the instruction parallely may affected by this.

According to the figure 2 time spent for parallel implementation is half the time spent for the serial implementation. Computer we used for the experiment has 2 cores and 4 hardware level threads. When implementing the serial implementation, it will only utilize one thread and sequentially carry out the multiplication process. But when we use the threading in parallel implementation, it will utilize all 4 cores and carry out the multiplication process. This will give a significant speedup over the serial implementation.

Even Though the number of thread available are 4 maximum speed up we gained is around 2. Time is not a good measurement of performance, but it can be seen that the underlying CPU is influencing the processing times. In parallel implementation it has to switch between multiple thread and also it need time to create, terminate and divide workloads. And this overhead will increase the time and reduce the desired speed up.

- Discuss your observations in detail while relating the observed behavior in graphs to the architecture of the CPU.

With increasing n, the serial and parallel execution times increase exponentially.

Figure 1 shows the time taken for serial and parallel execution. As known, matrix-matrix multiplication takes a time complexity of $O(n^3)$ and according to the graph we can see the exact behaviour such that the time taken for serial and parallel execution increases exponentially.

Parallel execution time increases relatively low

According to the hardware architecture processor contains two cores and four hardware threads. In the serial implementation all the execution happens sequentially and when the matrix size increases number of calculations increase under the complexity of $O(n^3)$ and only one thread is utilized to execute the multiplication. But in parallel implementation, the workload is divided into multiple cores and performs relatively faster. When the matrix size is small due to the overhead of thread creation, switching the actual performance gained will reduce. And when the matrix size increases the ratio between speed up gained by the parallel implementation and thread overhead increases. Hence, we can get the above behaviour.

In terms of cache of cache hierarchy, serial implementation try to do all the work in one thread, which makes if harder and more prone to cache misses and replacements, more frequently. But in parallel, this is less because it uses multiple cores and their caches in a parallel way.

When n is low serial and parallel execution times are almost same

Actual execution time include the thread creation, thread switching times and underline cache misses. When the matrix size is low this overhead may reduce the actual speed up. Hence the serial and parallel execution times are almost same. When the matrix is low thrashing may occur and that will also affect the speed.

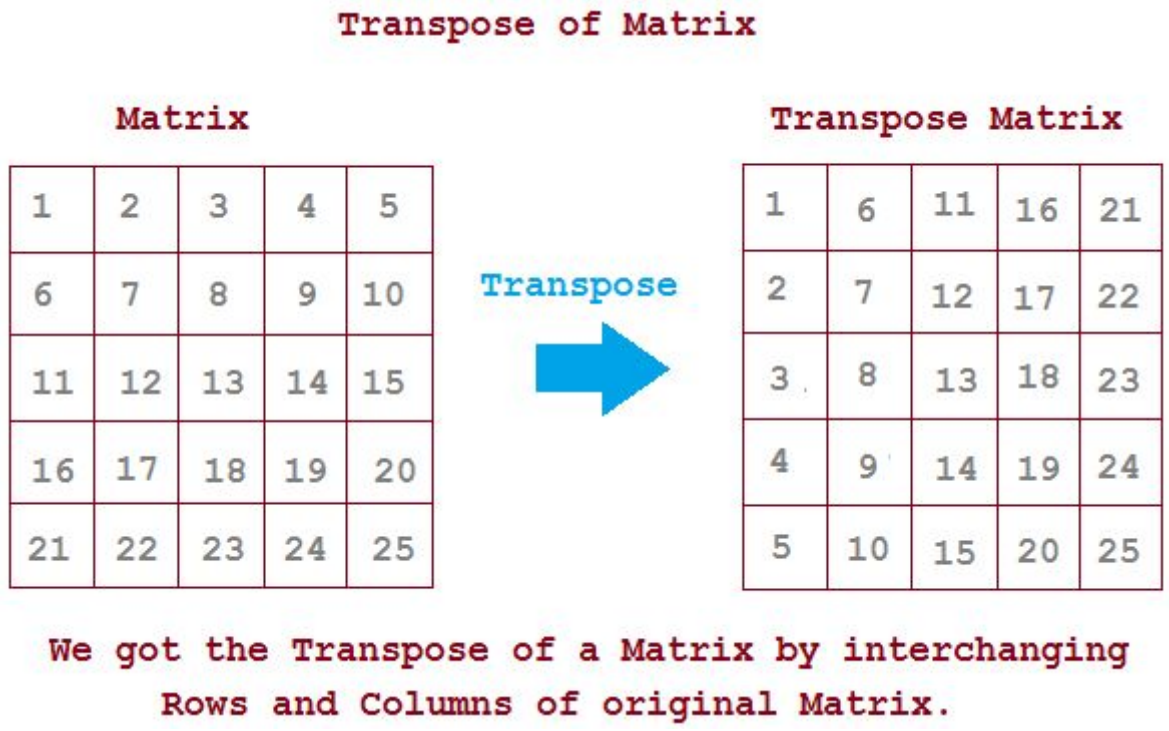
Fluctuation in the speed up with increasing n

This is not a very big fluctuation when we look into the actual execution times. These variations could occur due to threading, underlying CPU processes or even cache misses. But overall, Figure 2 depicts the gradual increase in the speed up of the computation.

6) Find out at least two ways to optimize matrix-matrix multiplication Briefly describe a possible technique(s) while giving references.

Following techniques are used to optimized the above tasks

1. Get the transpose of second matrix.

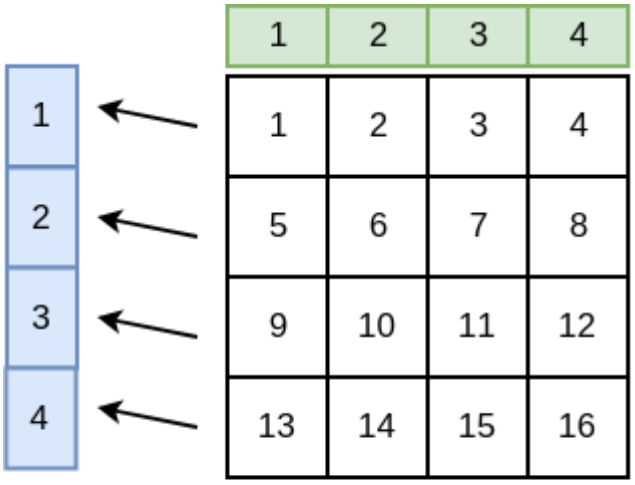


Normally the matrices are stored in the memory as 2D arrays and usually we use row major order. When it multiplying two matrices (A x B), the loops execute through the rows of A and columns of B. Then we cannot gain more cache level optimization form the execution because accessing matrix column wise will have more cache misses. Solution is to get the transpose of B before execution[1][1], [2].

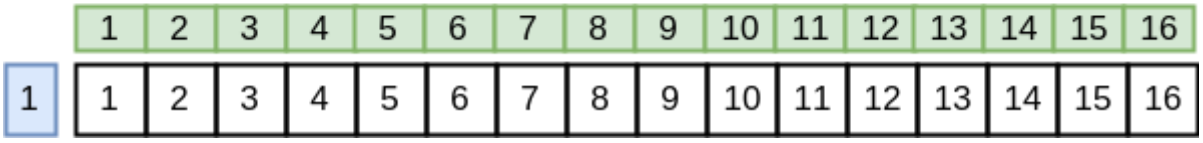
The algorithm is given below,

```
void mat_transpose(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            mat2_trans[j][i] = mat2[i][j];
        }
    }
}
```

2. Flat the matrices[1][2]



A[2][1] = B[4*2 + 1] = B[9]



When it considering 2D array, it is array of pointers to inner arrays. If we want to get a number of the matrix, the process have to go to the main pointer array and get the proper pointer to the inner array. To reduce the cost of finding from both this pointers, we take the full matrix as one dimension array.

For a 2D array, there is one pointer in the stack and it is connected to the array of pointers in the heap. That array of pointers contains another n number of arrays pointers. But in the 1D array we used here, has only one pointer in the stack and a one array in the heap.

3. Loop unrolling

```
for (int k = 0; k < n; k += BLOCK_SIZE) {
    sum += mat1_flat[i_n + k] * mat2_flat[j_n + k] +
        mat1_flat[i_n + k + 1] * mat2_flat[j_n + k + 1] +
        mat1_flat[i_n + k + 2] * mat2_flat[j_n + k + 2] +
        mat1_flat[i_n + k + 3] * mat2_flat[j_n + k + 3] +
        mat1_flat[i_n + k + 4] * mat2_flat[j_n + k + 4] +
        mat1_flat[i_n + k + 5] * mat2_flat[j_n + k + 5] +
        mat1_flat[i_n + k + 6] * mat2_flat[j_n + k + 6] +
        mat1_flat[i_n + k + 7] * mat2_flat[j_n + k + 7];
    }
}
```

As in the above code example, we do not want to iterate the for loop for whole values. We can do the calculation of set of iterations in one time in for loop. Then it can be ignored ‘k < n’ condition checking and ‘k += BLOCK_SIZE’.

4. g++ compiler O2 optimization

g++ compiler provide options to optimize the code. Since we use same matrix all the time we have to keep that in the stack for a long time. After some time stack memory is cleaned and using the O2 optimization will prevent this behaviour[3].

8) Optimized Solution

Optimized Output time for one sample

Matrix Dimension	Optimized Executing Time (s)
200	0.0061695
400	0.0191462
600	0.0616141
800	0.163321
1000	0.371622
1200	0.768014
1400	1.19588
1600	1.91789
1800	2.81099
2000	3.61776

Table 6: Optimized execution average time

- Collect sufficient number of samples for each configuration of n, while making sure your performance results are within an accuracy of ±5% and 95% confidence level.

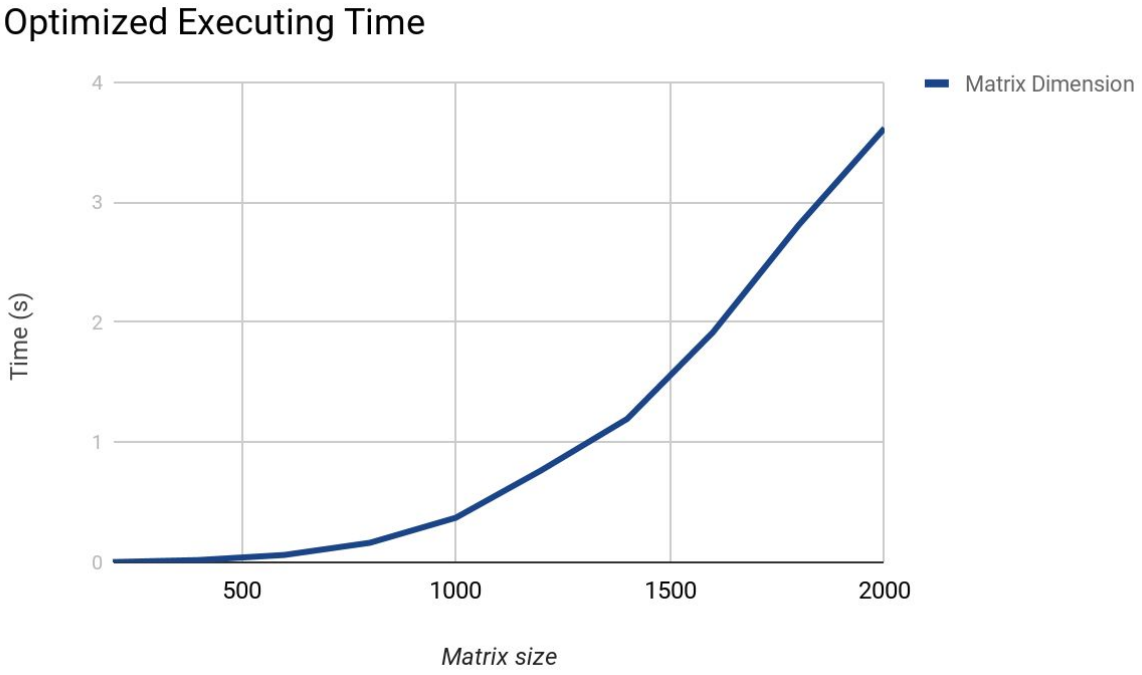
Dimension	Used Sample size	Sample mean	Sample standard deviation	Required sample size
200	20	0.00244393	0.00105846	289
400	20	0.0163341	0.000695109	3
600	20	0.0574871	0.0016285	2
800	20	0.166454	0.00544605	2
1000	20	0.469363	0.02191	4
1200	20	0.807464	0.0341266	3
1400	20	1.35063	0.045307	2
1600	20	1.95183	0.0901076	4
1800	20	2.72415	0.0902231	2
2000	20	3.81191	0.122203	2

Table 7:Optimized execution time with required sample size

Average time of executing for the required number of samples for 95% confidence level.

Matrix Dimension	Optimized Executing Time (s)
200	0.00227741
400	0.0191462
600	0.0616141
800	0.163321
1000	0.371622
1200	0.768014
1400	1.19588
1600	1.91789
1800	2.81099
2000	3.61776

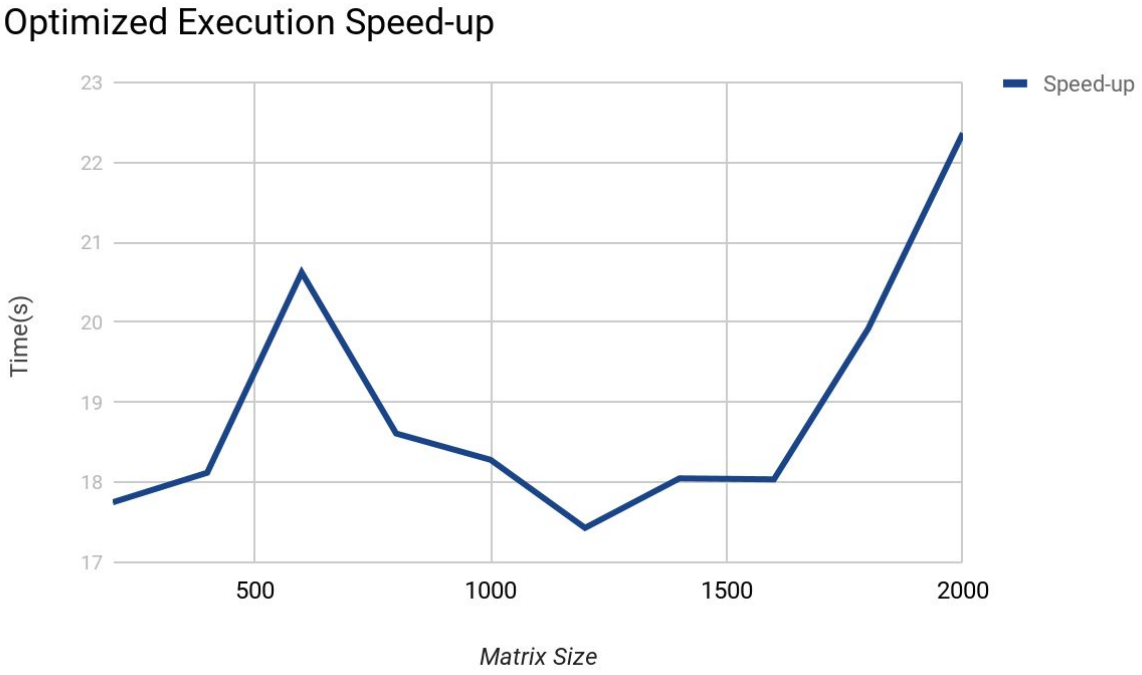
Table 8: Execution Time for optimized.



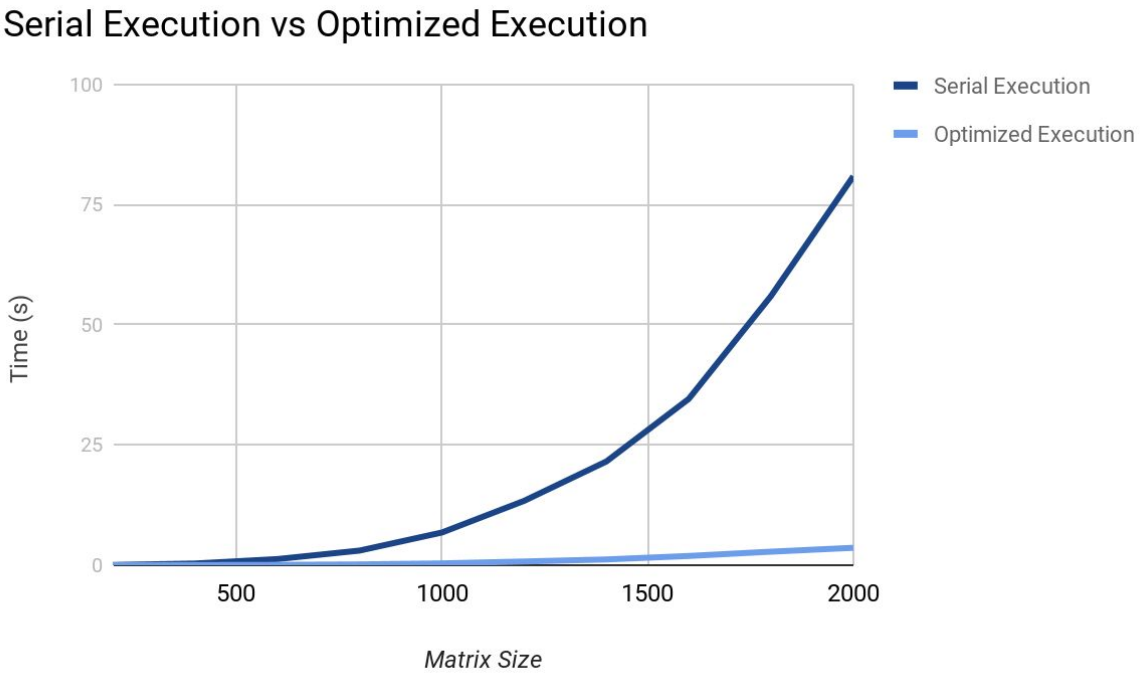
Graph 3: Speed up of optimized solution

Dimension	Avg time for serial	Avg time for optimized execution	Speed up
200	0.040426	0.00227741	17.75086612
400	0.346936	0.0191462	18.12035809
600	1.27073	0.0616141	20.62401301
800	3.03956	0.163321	18.61095634
1000	6.79399	0.371622	18.28199084
1200	13.3873	0.768014	17.43106245
1400	21.5861	1.19588	18.05038967
1600	34.5948	1.91789	18.03794795
1800	56.0104	2.81099	19.92550667
2000	80.9087	3.61776	22.3643083

Table 9: Speedup between optimized and serial execution time



Graph 4: Speed-up vs matrix size



Graph 5: Speed-up of optimized and serial execution.

Observation and discussion

Graph 5 shows the average time take for optimized implementation. Usually the matrix multiplication algorithms has faster rate than O(n3) due to improved memory use by blocking.

Observation 1:
The CPU architecture of the considered machine has 2 cores and 4 threads. In serial implementation, it has used only one core and it will utilized to do the whole task by using only one processor. But when we using multithreading environment, it can utilize both cores and 4 threads and dot eh process parallely.

Processors will have to do more works when it increase the matrix size. to reduce the time which is taken to switch data between two threads or cache we used some locallity techniques.

Observation 2:
In the speed-up graphs, it shows for the small matrix sizes the speed-up is less but it is highly increasing when it comes to larger matrix sizes.

This is because the size of the problem is comparatively small for multiple threads to handle and the time taken to initiate, schedule and terminate threads becomes more significant. As n increases, this time taken to schedule becomes less significant as more work is done in parallel by the threads.

Observation 3:
The actual execution time has not a big time difference and when it consider the multithreading environment we cannot guaranteed about the time because of some number of factors like cache misses, page faults. but in Graphs 3 we can see the time is gradually increasing.

[1] X. Fan, “Optimize Your Code: Matrix Multiplication,” *Van’s House*. [Online]. Available: https://blogs.msdn.microsoft.com/xiangfan/2009/04/28/optimize-your-code-matrix-multiplication/. [Accessed: 23-Feb-2018].

[2] “CMSC411 PROJECT: Cache, Matrix Multiplication, and Vector... Presented by: Presented by: Hongqing Liu, Stacy Weng, and Wei Sun.” [Online]. Available: https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/cache/matrix.html. [Accessed: 23-Feb-2018].

[3] “Using the GNU Compiler Collection (GCC): Optimize Options.” [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Accessed: 23-Feb-2018].