

CS-4532 Concurrent Programming

Take Home Lab 3 and 4

Lab Members:

- V. G. Mallwaarachchi - 130366E
- K. M. Sugathadasa - 130581H

Github Code URI

- <https://github.com/Vini2/ParallelMatrixMultiplicationUsingOpenMP>

Task 4

a)

Time taken for each execution - with sample size of 1

n (nxn matrix dimension)	Serial (seconds)	Parallel (seconds)
200	0.032000	0.025000
400	0.216000	0.101000
600	1.056000	0.434000
800	2.805000	1.204000
1000	5.951000	2.614000
1200	10.895000	4.713000
1400	17.772000	7.823000
1600	27.524000	12.114000
1800	50.312000	17.578000
2000	68.234000	24.645000

Table 1: Serial Matrix Matrix Multiplication Execution Times

b)

We used a sample size of 100 initially and calculated the sample mean and sample deviation as **m** and **s** respectively.

With an accuracy level of 5% and 95% confidence level, we applied the sample mean and sample deviation to the following equation and obtained the required number of samples. This proved us that, the sample size of 100 was sufficient the get an accuracy of 5% and a confidence level of 95%.

$z = 1.960$

s = sample sd,

r = required accuracy

m = sample mean

n = required sample size

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

This table is for Serial implementation values to calculate required sample size

SERIAL IMPLEMENTATION				
n (nxn matrix dimension)	Used sample size	Sample mean	Sample standard deviation	Required sample size
200	100	0.030000	0.223036	65
400	100	0.273667	0.572819	47
600	100	1.022333	0.740052	21
800	100	2.736333	1.023263	15
1000	100	6.393333	1.211551	9
1200	100	11.091667	1.504528	8
1400	100	18.136333	1.360385	4
1600	100	29.290667	0.818951	1
1800	100	43.936000	1.004358	1
2000	100	60.185000	0.959791	1

Table 2: Serial Matrix Matrix Multiplication with samples to calculate required sample size

This table is for Parallel implementation values to calculate required sample size

PARALLEL IMPLEMENTATION				
n (nxn matrix dimension)	Used sample size	Sample mean	Sample standard deviation	Required sample size
200	100	0.014040	0.172417	83
400	100	0.116220	0.403812	55
600	100	0.447100	0.575120	29
800	100	1.218730	0.768577	19
1000	100	2.589360	0.726939	8
1200	100	4.705170	0.0.60007	3
1400	100	7.901820	0.448973	1
1600	100	12.151210	0.556758	1
1800	100	17.891220	0.477707	1
2000	100	25.002300	0.357159	1

Table 3: Parallel Matrix Matrix Multiplication with samples to calculate required sample size

The required sample size for every n, is less than 100. Hence a sample size of 100 is sufficient to provide results with 5% accuracy level and 95% confidence level.

c)

When plotting the matrix matrix multiplication time, against increasing n, use the following table.

n (nxn matrix dimension)	Average Time (Serial)	Average Time (Parallel)
200	0.030000	0.014040
400	0.273667	0.116220
600	1.022333	0.447100
800	2.736333	1.218730
1000	6.393333	2.589360
1200	11.091667	4.705170

1400	18.136333	7.901820
1600	29.290667	12.151210
1800	43.936000	17.891220
2000	60.185000	25.002300

Table 4: Serial and Parallel Matrix Matrix Multiplication with average execution times

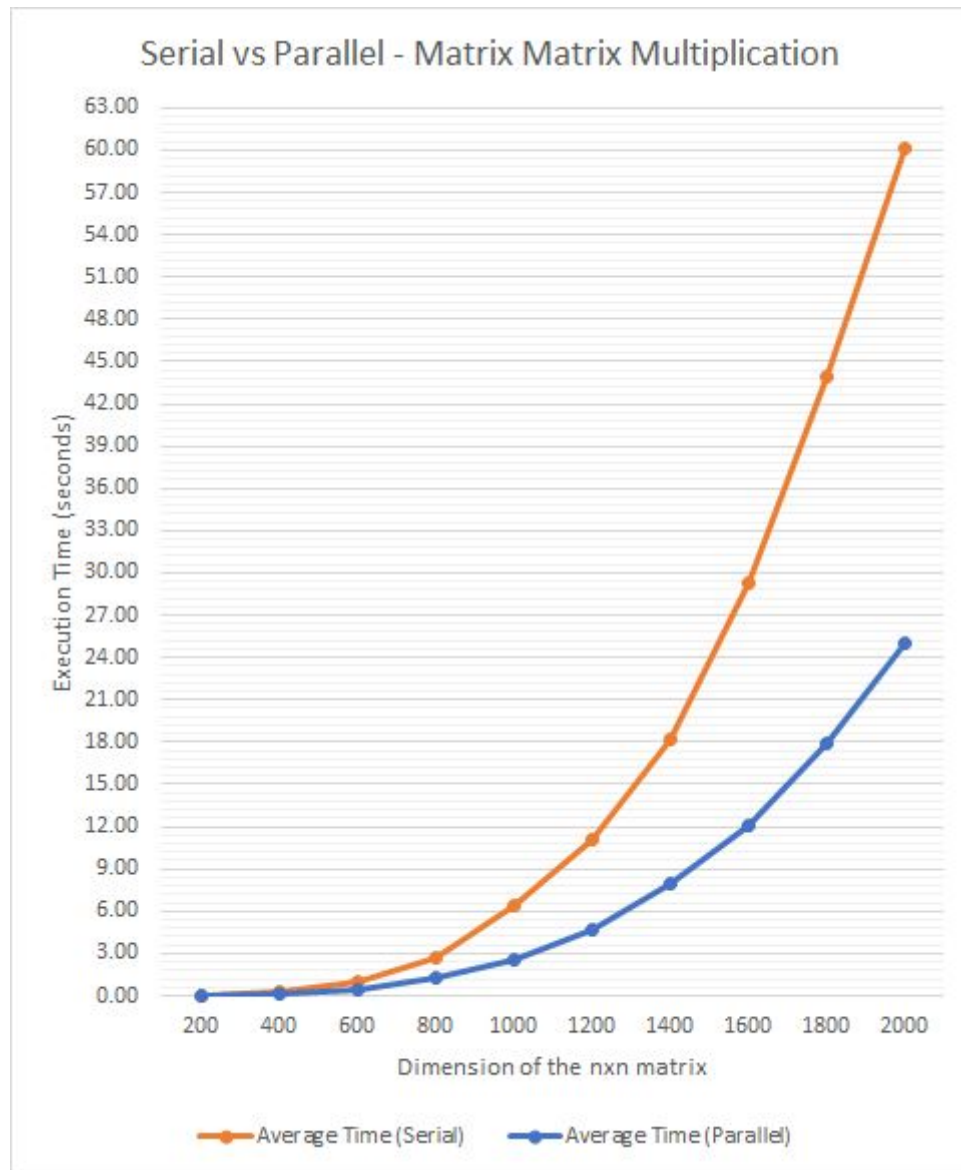


Figure 1: Graph of serial and parallel matrix-matrix multiplication time against increasing n

d)

Next, we plot the speed up we gained from serial to parallel, against increasing n.

$$\text{Speed up} = \frac{\text{serial execution time}}{\text{parallel execution time}}$$

n (nxn matrix dimension)	Average Time (Serial) - Seconds	Average Time (Parallel) - Seconds	Speed up
200	0.030000	0.014040	2.136752137
400	0.273667	0.116220	2.354732404
600	1.022333	0.447100	2.286586893
800	2.736333	1.218730	2.245233153
1000	6.393333	2.589360	2.46907846
1200	11.091667	4.705170	2.357336079
1400	18.136333	7.901820	2.295209585
1600	29.290667	12.151210	2.410514426
1800	43.936000	17.891220	2.455729682
2000	60.185000	25.002300	2.40717854

Table 5: Parallel vs Serial Matrix Matrix Multiplication Speedup

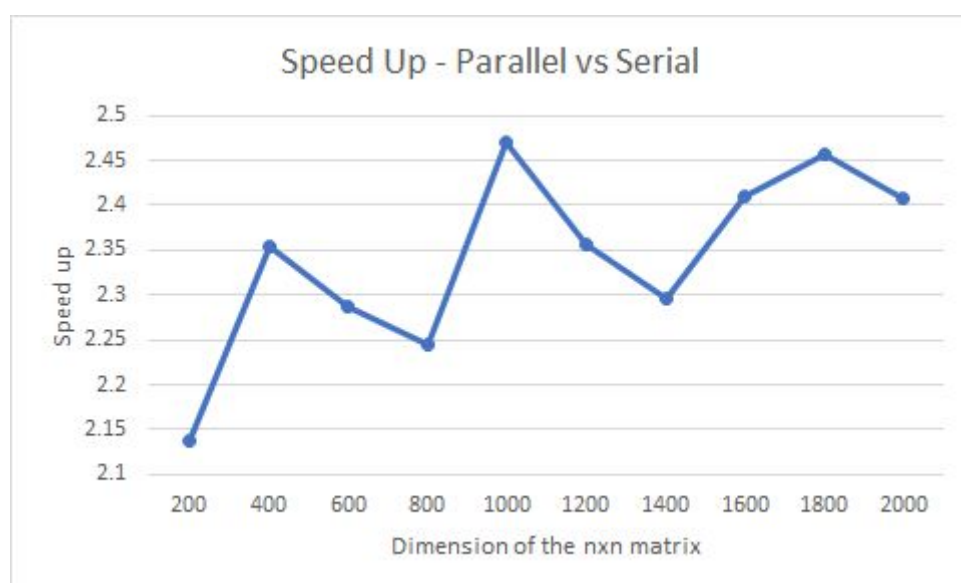


Figure 2: Graph of serial vs. parallel speed up against increasing n

Task 5

a)

- CPU model: Intel Core i7 6600U
- No of Cores: 2
- No of hardware level threads: 4


Processor					
Name	Intel Core i7 6600U				
Code Name	Skylake-U/Y	Max TDP	15.0 W		
Package	Socket 1168 BGA				
Technology	14 nm	Core VID	1.006 V		
					
Specification	Intel® Core™ i7-6500U CPU @ 2.50GHz				
Family	6	Model	E	Stepping	3
Ext. Family	6	Ext. Model	4E	Revision	D0/K0/K1
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3				
Caches (Core #0)					
Core Speed	2994.87 MHz				
Multiplier	x 30.0 (4 - 31)				
Bus Speed	99.78 MHz				
Rated FSB					
Cache					
L1 Data	2 x 32 KBytes		8-way		
L1 Inst.	2 x 32 KBytes		8-way		
Level 2	2 x 256 KBytes		4-way		
Level 3	4 MBytes		16-way		
Selection <input type="text" value="Socket #1"/>					
Cores		2	Threads		4

Figure 3: CPU model, no of cores, no of hardware-level threads

- Cache Hierarchy:

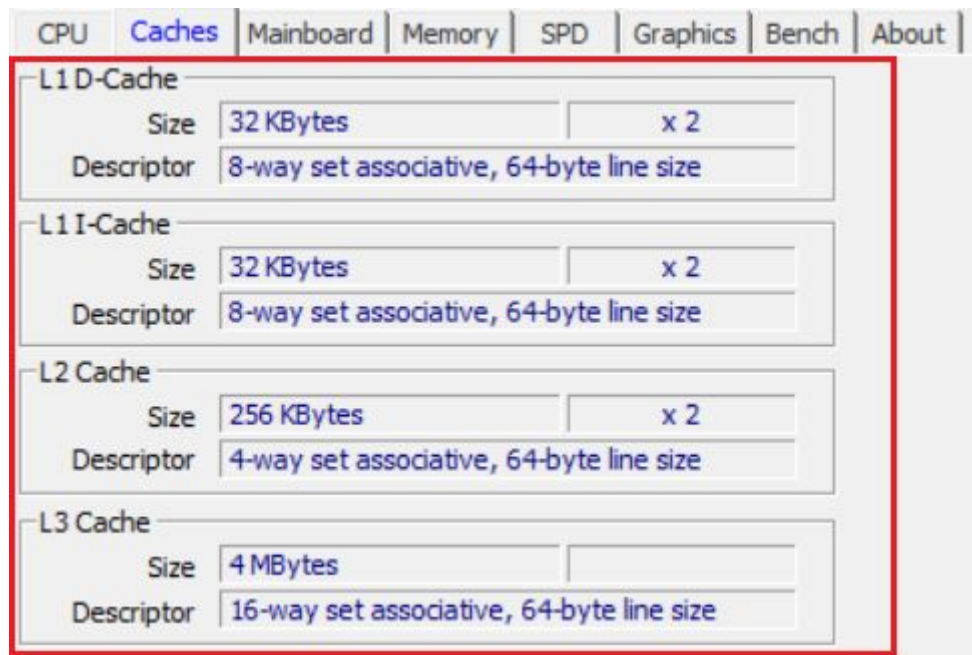


Figure 4: Cache details

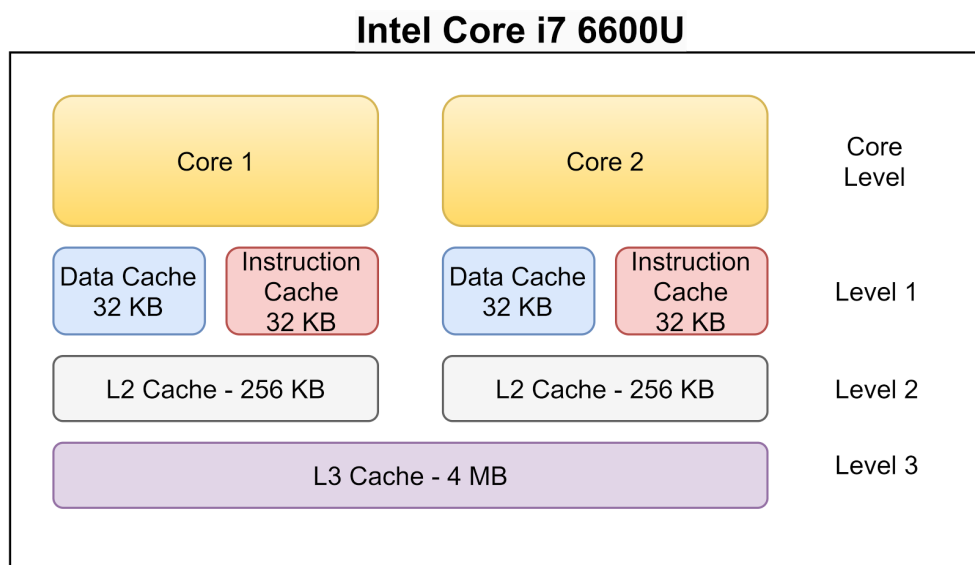


Figure 5: Cache hierarchy

- Memory: DDR3 8088 MB (Dual Channel)
- Hardware optimizations: Cache locality (Spatial locality), Multi-level cache (L1, L2, L3)

b)

Justification for gained speed up

According to the speed up given in figure 2, we can see a gradual increase in the speed up when comparing the parallel vs serial matrix matrix multiplication algorithms. There are minor variations in the speed up against increasing n , but what should be considered is the gradual increase in speed up.

As shown in figure 3, this CPU architecture has 2 cores and four threads. When implementing serial multiplication, it will only utilize one of the threads and sequentially carry out the multiplication process. But when we use threading in parallel multiplication, it can utilize both cores with 4 threads and do the process parallelly. This will give a significant speedup over the serial multiplication process.

Then The problem arises as to why there is no 4 times speed up if there are 4 threads in the given CPU architecture. Time is not a good measure of performance, but it can give be seen that the underlying CPU is influencing the processing times. In parallel multiplication, it has the switch between threads, write to shared resources, initialize and terminate threads, and dividing workload accordingly to threads, takes quite a bit of time, which causes the parallel multiplication, not to give the ideal speed up we expect.

c)

Discussion on observations

Figure 1 shows the average times taken for matrix matrix multiplications for both serial and parallel. As known, the matrix matrix multiplication takes a time complexity of $O(n^3)$ and with increasing n , the time taken for processing increases exponentially.

Observation 1: With increasing n , the serial processing time increases exponentially, where the parallel processing time increase is relatively low.

If we look at the hardware architecture, this computer has 2 cores and 4 threads. So in serial multiplication, it will only utilize one core and one thread. Due to this reason, when n is increasing, the number of calculations needed increases and the time grows exponentially with the complexity of $O(n^3)$. But in parallel multiplication, this workload is divided among the 4 threads in the computer, and performs relatively faster than the serial process.

In terms of the cache hierarchy, the serial multiplication tries to do all the work using a single core and its cache, which makes it harder and more prone to cache misses and replacements, more frequently. But in parallel, this is less because it uses both cores and their caches in a parallel way. So the execution time in parallel is relatively faster than the serial one.

Observation 2: Initially, when n is low, Serial and parallel execution times, are almost similar

When the workload is low, the serial process performs as well as the parallel process. This is mainly because, the parallel multiplication tries to divide the small workload, among threads and this takes quite a bit time to initiate and terminate threads, where it makes the parallel process no different from the serial one. But as in observation one, when n increases, parallel performs well.

As depicted in figure 2, the speed up when n is low, is comparatively low. This is mainly because parallel multiplication does not perform significantly faster when n is small, due to switching of threads and processing, whereas serial multiplication may perform with a similar speed when n is small, due to the small number of calculations. But as n increases, the workload gets tedious and serial multiplication gets overwhelmed on one thread (core) and tries to finish the calculation. But parallel performs better with increasing n , as the workload gets separated amongst threads for parallel processing.

Observation 3: Figure 2 shows some fluctuations 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.

Task 6

Following are some methods that we can use to optimize matrix multiplication.

Method 1: Transposing the second matrix

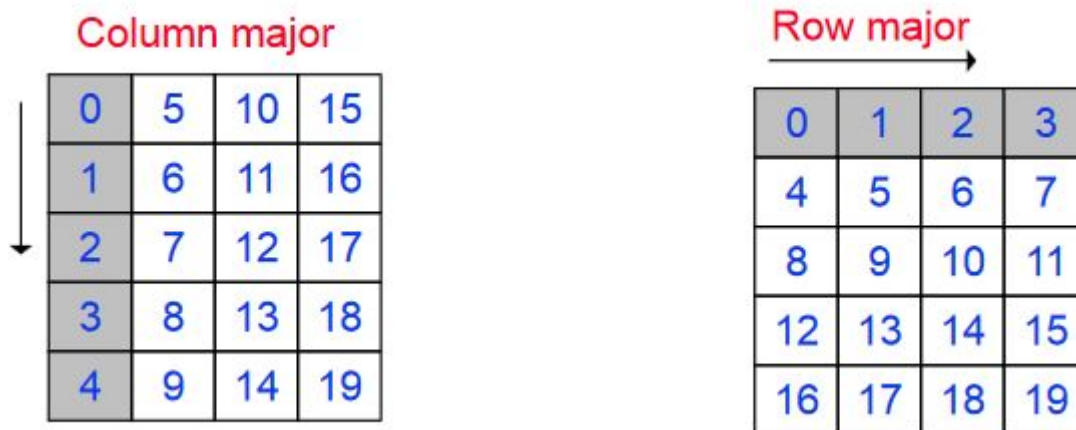


Figure 6: Column major and row major matrices

Image Source: <http://web.cs.ucdavis.edu/~bai/ECS231/optmatmul.pdf>

When storing matrices in programs, the general practice is to store in row major order.

When we consider the multiplication of two matrices A and B, A will be accessed in row major order while B will be accessed in column major order. Accessing elements in row major order takes time as the elements are not located close by. We transpose the matrix B so that both A and B can be accessed in row major order which is in sequential order. This method makes use of the spatial locality of cache memory.

The algorithm is given below.

```
Transpose(B)
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0;
        for (int k = 0; k < n; k++) {
            sum += A[i][k] * B[j][k];
        }
        C[i][j] = sum;
    }
}
```

References: [1] [2]

Method 2: Blocked (Tiled) Matrix Multiplication

When it comes to multiplying large matrices, the entire matrix may not fit into memory. Hence we break the matrix into smaller chunks of elements, known as **blocks**, which can be moved quickly into memory.

The algorithm used is given below.

```

for (i0 = 0; i0 < n; i0 = i0 + blockSize){
    for (j0 = 0; j0 < n; j0 = j0 + blockSize){
        for (k0 = 0; k0 < n; k0 = k0 + blockSize){
            for (i = i0; i < min(i0+blockSize, n); i++){
                for (j = j0; j < min(j0+blockSize, n); j++){
                    for (k = k0; k < min(k0+blockSize, n); k++){
                        C[i][j] += A[i][k] * B[k][j]
                    }
                }
            }
        }
    }
}

```

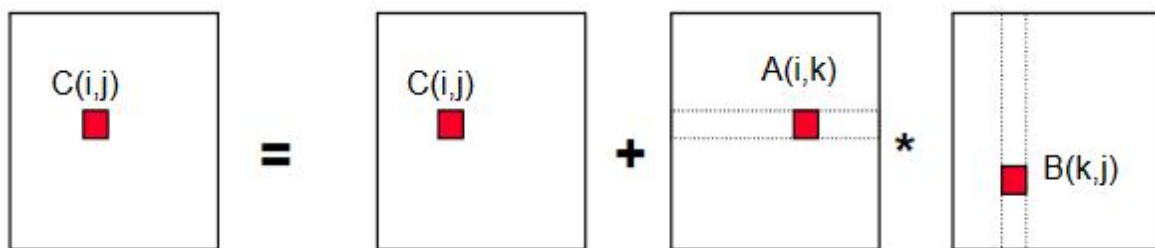


Figure 7: Blocks of matrices

Image Source: <http://web.cs.ucdavis.edu/~bai/ECS231/optmatmul.pdf>

Here block size should be chosen wisely so that the blocks of matrix A, B and C can be fitted in fast memory or Cache at the same time.

References: [2] [3] [4]

Task 7

a)

Execution time for each dimension, on the optimized algorithm

n (nxn matrix dimension)	Optimized Time (Seconds)
200	0.000000
400	0.055000
600	0.164000
800	0.198000
1000	0.300000
1200	0.398000
1400	0.496000
1600	0.681000
1800	0.941000
2000	1.255000

Table 6: Parallel Optimized Matrix Matrix Multiplication Execution Times

b)

We used a sample size of 100, and obtained that it is sufficient to get performance results within an accuracy of 5 % and 95% confidence level.

With an accuracy level of 5% and 95% confidence level, we applied the sample mean and sample deviation to the following equation and obtained the sufficient number of samples. This proved us that, the sample size of 100 was sufficient the get an accuracy of 5% and a confidence level of 95%.

$$Z = 1.960$$

s = sample sd,

r = required accuracy

m = sample mean

n = required sample size

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

OPTIMIZED IMPLEMENTATION				
n (nxn matrix dimension)	Used sample size	Sample mean	Sample standard deviation	Required sample size
200	100	0.006290	0.107485	72
400	100	0.048890	0.259516	54
600	100	0.157020	0.451980	51
800	100	0.200720	0.033978	44
1000	100	0.394770	0.301058	9
1200	100	0.423000	0.232280	5
1400	100	0.572020	0.114600	1
1600	100	0.698500	0.188780	2
1800	100	1.122000	0.169182	1
2000	100	1.437130	0.191471	1

Table 7: Parallel Optimized Matrix Matrix Multiplication with samples to calculate required sample size

The required sample size for every n is less than 100. Hence a sample size of 100 is sufficient to provide results with 5% accuracy level and 95% confidence level.

c)

Plotting the matrix matrix multiplication time, against increasing n. Use the following table to plot it

n (nxn matrix dimension)	Average Time (Optimized)
200	0.006290
400	0.048890
600	0.157020
800	0.200720
1000	0.394770
1200	0.423000

1400	0.572020
1600	0.698500
1800	1.122000
2000	1.437130

Table 8: Parallel Optimized Matrix Matrix Multiplication with Average Execution Times

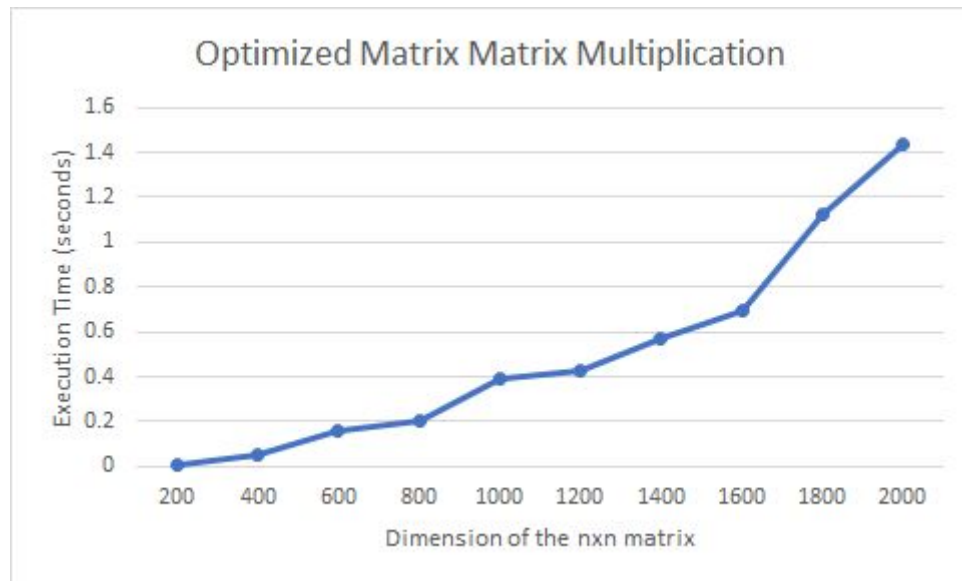


Figure 8: Graph of optimized matrix-matrix multiplication time against increasing n

d)

Next, we plot the speed up we gained from serial to optimized, against increasing n.

$$Speed\ up = \frac{serial\ execution\ time}{optimized\ execution\ time}$$

n (nxn matrix dimension)	Average Time (Serial) - Seconds	Average Time (Optimized) - Seconds	Speed up
200	0.030000	0.006290	4.769475358
400	0.273667	0.048890	5.597606873
600	1.022333	0.157020	6.510845752
800	2.736333	0.200720	13.63258768

1000	6.393333	0.394770	16.19508321
1200	11.091667	0.423000	26.22143499
1400	18.136333	0.572020	31.70576728
1600	29.290667	0.698500	41.93366786
1800	43.936000	1.122000	39.15864528
2000	60.185000	1.437130	41.74987649

Table 9: Parallel Optimized vs Serial Matrix Matrix Multiplication Speedup

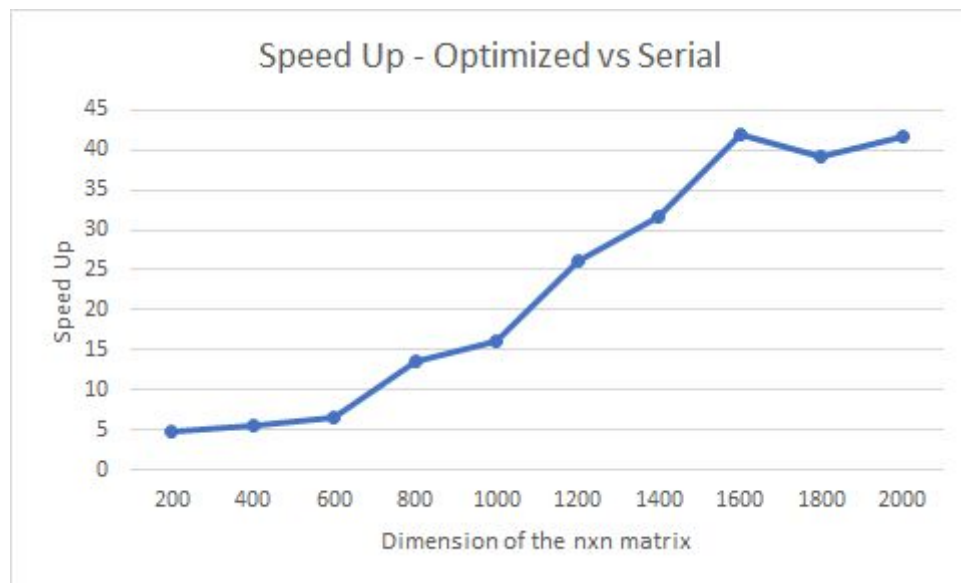


Figure 9: Graph of serial vs. optimized speed up against increasing n

e)

Optimization Techniques Used

The technique used to optimize parallel for matrix matrix multiplication is: **blocked (tiled) matrix multiplication**. Here the matrices are broken into small chunks called *blocks* and calculations are done on blocks parallelly (as described in Task 6).

This method allows large matrices to be broken down into pieces of manageable size that will fit in memory for ease of computation. Having such manageable sized pieces allows to reduce the number of shifts into and out of memory. It also provides better spatial locality so that the cache can be used efficiently, in order to access data quickly. This provides improved performance in parallel codes due to better data reuse.

Observations and Discussion

Figure 8 shows the average times taken for matrix matrix multiplications for optimized implementation. It is known that the matrix matrix multiplication using block algorithms has a faster rate than $O(n^3)$ due to improved memory use by blocking.

Observation 1: According to the running times shown in figure 8, we can see a gradual increase in the running time of optimized matrix matrix multiplication algorithm.

The block size has been defined as,

$$\text{Block size} = \frac{\text{Matrix size } (n)}{\text{Number of threads}}$$

As the matrix size increases, the block size increases as well, because we have kept the number of a threads constant. As the block size increases, threads have to perform more work for calculations and thus increasing the running time.

Observation 2: Figure 9 shows a gradual increase in speed up of serial vs optimized matrix matrix multiplication algorithms.

As described in Task 5, the CPU architecture of the considered machine has 2 cores and 4 threads. When implementing serial multiplication, it will only utilize one of the threads and sequentially carry out the multiplication process. However when we use threading in optimized multiplication, it can utilize both cores with 4 threads and do the process parallelly. This will give a significant speedup of more than 4 times over the serial multiplication process.

As the matrix size increases, more work is done in parallel, reducing the time taken to shift data in and out of memory. This results in an increased speed up. But serial will keep on processing on a single thread, making it to fall behind due to increasing workload, with increasing n.

Observation 3: According to figure 9, it can be seen that for lower values of n such as 200 and 400, the speed up is not very significant when compared to the speed up of higher values of n such as 1000 onwards.

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 4: There are minor variations in the speed up against 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 8 depicts the gradual increase in the speed up of the computation. However when considering the overall picture, there is a gradual increase in speed up.

References

- [1] Optimize Your Code: Matrix Multiplication - <https://blogs.msdn.microsoft.com/xiangfan/2009/04/28/optimize-your-code-matrix-multiplicati on/>
- [2] Uniprocessor Optimization of Matrix Multiplications and BLAS - <http://web.cs.ucdavis.edu/~bai/ECS231/optmatmul.pdf>
- [3] Block algorithms: Matrix Multiplication as an Example - <http://www.netlib.org/utk/papers/autoblock/node2.html>
- [4] Strassen algorithm - https://en.wikipedia.org/wiki/Strassen_algorithm