CSCE 735 Major Project

Parallelizing Strassen's Matrix Multiplication Algorithm

Name: Pranav Anantharam
UIN: 734003886

1. I have chosen the following strategy for the parallel implementation of Strassen's Matrix multiplication algorithm: **Develop a shared-memory code using OpenMP.**

The code is submitted separately as a zip file on Canvas.

2.

The parallel implementation of Strassen's Matrix multiplication algorithm is contained in the file 'major_project.cpp'.
First, we receive the inputs for the matrix size, number of levels of recursion (terminal matrix size) and number of threads. Next, we initialize matrices A, B, Result and Test and fill in the matrices A and B with random integer values. Using Strassen's algorithm, we compute the resultant matrix and store it in Result while keeping track of the execution time. In the 'strassen_mul' function, we check if the matrix size is less than or equal to the terminal matrix size. If true, then we perform the traditional matrix multiplication algorithm and compute the result. If false, then we divide the matrix into submatrices and recursively call the 'strassen_mul' and calculate the product of the submatrices. Finally, we compare the resultant matrix with the test matrix (computed using traditional matrix multiplication algorithm) and calculate the error count and display the results.

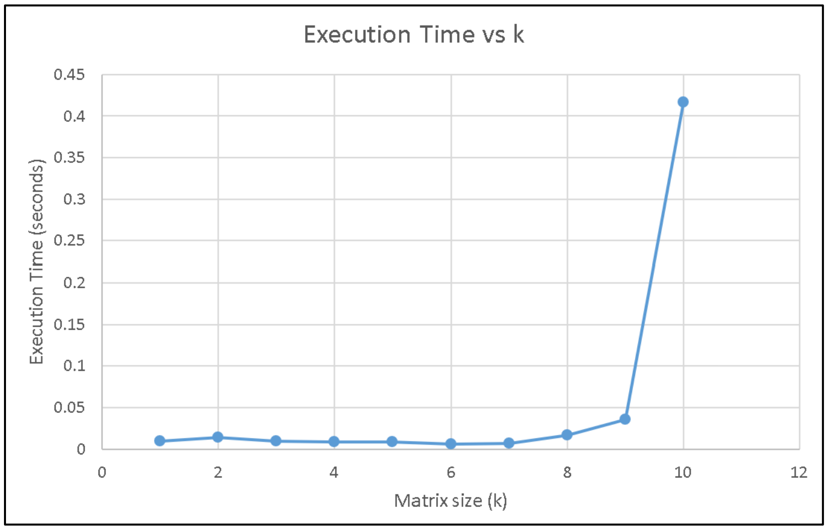
The following experiments were performed, and the below observations were noted:
The execution time (seconds) refers to the time taken to multiply two matrices of size **n x n** using input **k** and specified number of threads with a terminal matrix of size **k - k'**.

a) Varying Matrix sizes: k
(k' and number of threads are constant)

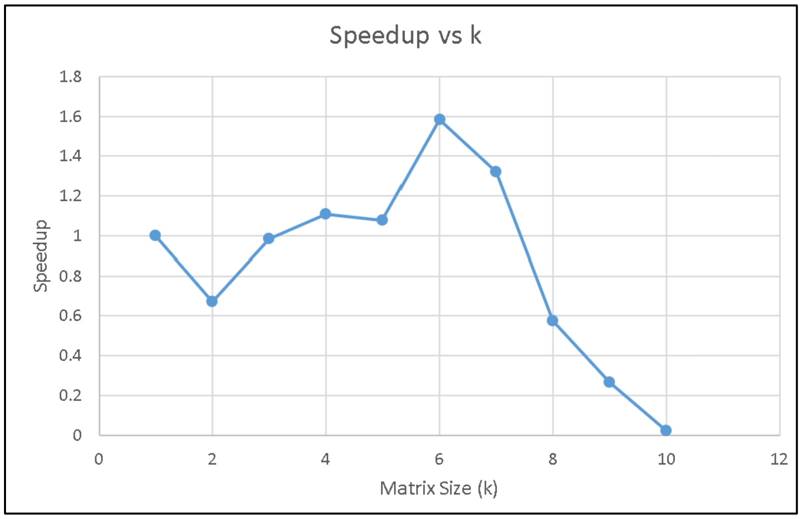
Spreadsheet Calculations:

Matrix Size	k	k'	Number of Threads (p)	Execution Time (in seconds)	Errors	Speedup	Efficiency
2	1	1	32	0.009382	0	1	0.03125
4	2	1	32	0.014006	0	0.669855776	0.020932993
8	3	1	32	0.009549	0	0.982511258	0.030703477
16	4	1	32	0.008477	0	1.106759467	0.034586233
32	5	1	32	0.00872	0	1.075917431	0.03362242
64	6	1	32	0.005933	0	1.581324794	0.0494164
128	7	1	32	0.007114	0	1.318807984	0.04121275
256	8	1	32	0.016364	0	0.573331704	0.017916616
512	9	1	32	0.035526	0	0.264088273	0.008252759
1024	10	1	32	0.416331	0	0.022534954	0.000704217

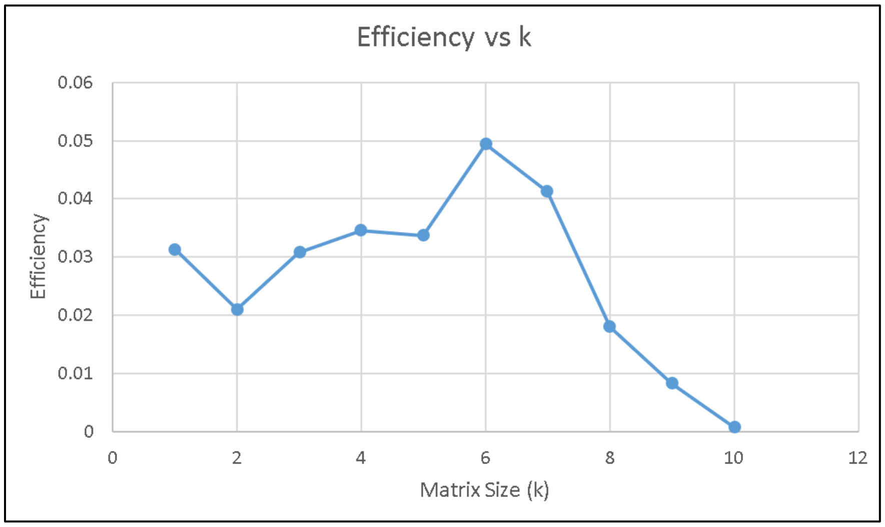
Execution Time vs Matrix Size (k):



Speedup vs Matrix Size (k):



Efficiency vs Matrix Size (k):



We observe that the execution time increases as the value of **k** increases. However, the rate of increase is not constant. Minor fluctuations are observed followed by a significant increase in execution when **k = 9**.

The speedup and efficiency plots display the same trends. The speedup values fluctuate between **0.6** and **1.2** until reaching a peak of **1.6** when **k = 6**, which is then followed by a steep decline.

The efficiency values also fluctuate between **0.02** and **0.04** until reaching peak efficiency of **0.05** at **k = 6**, which is then followed by a steep decline. No errors were observed in any of the experiments conducted.

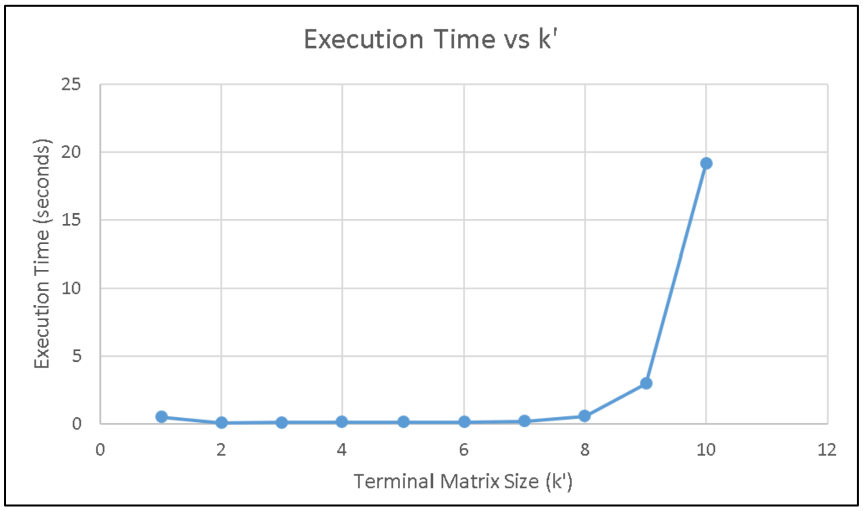
b) Varying number of levels of recursion / Terminal Matrix Size: k'

(k and number of threads are constant)
(We are indirectly defining the terminal matrix size)

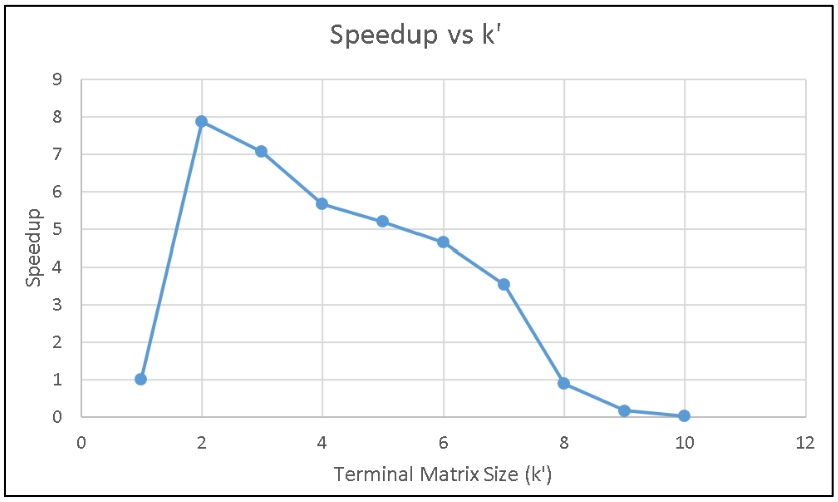
Spreadsheet Calculations:

Matrix Size	k	k'	Number of Threads (p)	Execution Time (in seconds)	Errors	Speedup	Efficiency
1024	10	1	32	0.456325	0	1	0.03125
1024	10	2	32	0.057963	0	7.87269465	0.246021708
1024	10	3	32	0.064578	0	7.066260956	0.220820655
1024	10	4	32	0.080509	0	5.667999851	0.177124995
1024	10	5	32	0.087748	0	5.200403428	0.162512607
1024	10	6	32	0.098394	0	4.637731976	0.144929124
1024	10	7	32	0.129312	0	3.528868164	0.11027713
1024	10	8	32	0.514974	0	0.886112697	0.027691022
1024	10	9	32	2.910474	0	0.156787176	0.004899599
1024	10	10	32	19.094804	0	0.023897862	0.000746808

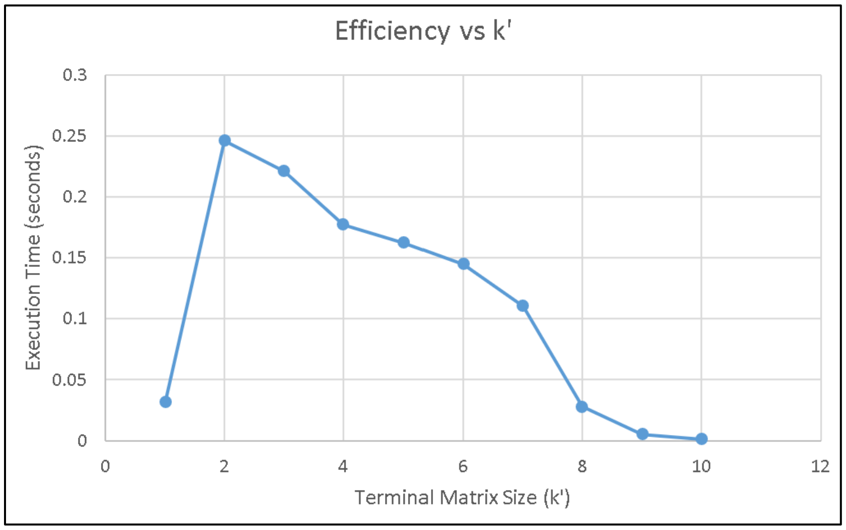
Execution Time vs k':



Speedup vs k':



Efficiency vs k':



We observe that the value of **k'** has an impact on the execution time for matrix multiplication. When **k' = 2** the execution time is the lowest and beyond this point the execution time slowly increases until we reach **k'= 10**, where we observe a significant increase in the execution time (almost **335** times the value observed at **k' = 2**).

Speedup and efficiency plots follow similar trends, where we observe an initial steep increase in speedup and efficiency when **k' = 2**, followed by a gradual decline. No errors were observed in any of the experiments conducted.

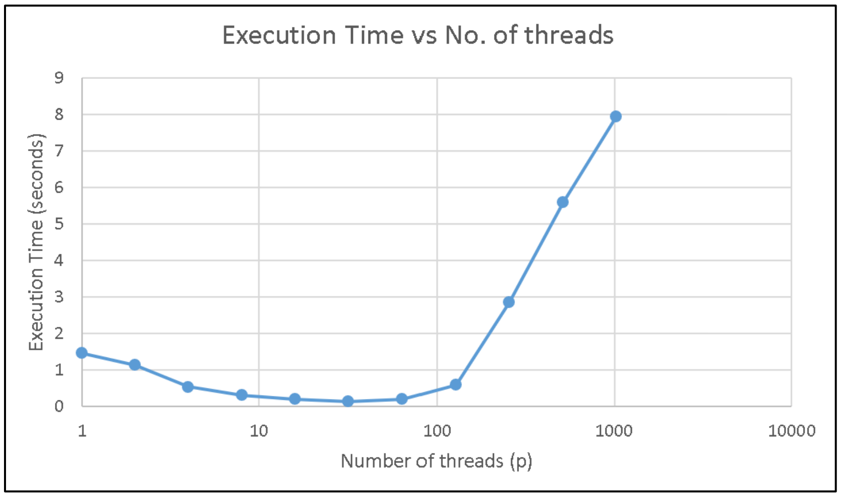
c) Varying number of threads: num_threads

(k and k’ are constant)

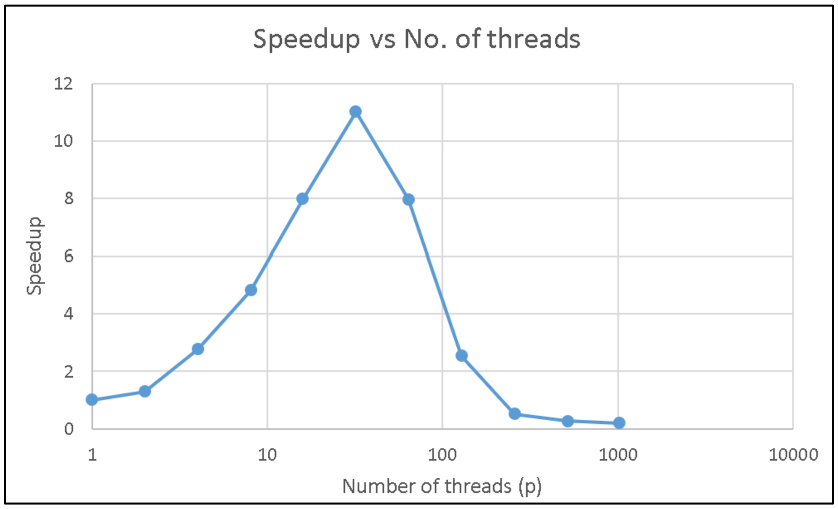
Spreadsheet Calculations:

Matrix Size	k	k'	Number of Threads (p)	Execution Time (in seconds)	Errors	Speedup	Efficiency
1024	10	7	1	1.450281	0	1	1
1024	10	7	2	1.12497	0	1.289173045	0.644586522
1024	10	7	4	0.522318	0	2.776624585	0.694156146
1024	10	7	8	0.301303	0	4.813363956	0.601670494
1024	10	7	16	0.182156	0	7.961752564	0.497609535
1024	10	7	32	0.131591	0	11.02112606	0.34441019
1024	10	7	64	0.182194	0	7.96009199	0.124376437
1024	10	7	128	0.571147	0	2.539242962	0.019837836
1024	10	7	256	2.830284	0	0.512415362	0.002001623
1024	10	7	512	5.575996	0	0.260093623	0.000507995
1024	10	7	1024	7.931175	0	0.182858278	0.000178573

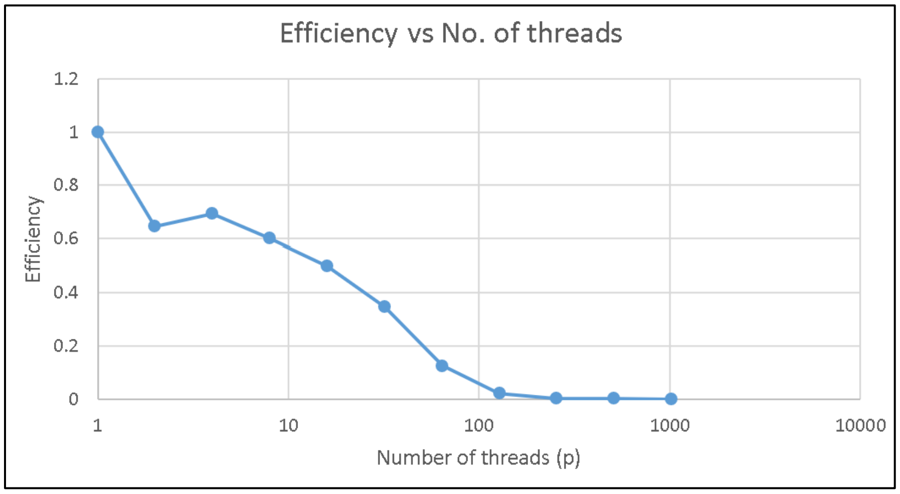
Execution Time vs No. of threads (p):



Speedup vs No. of threads (p):



Efficiency vs No. of threads (p):



We observe that the number of threads used for parallel processing has a significant impact on the execution time for matrix multiplication. As the number of threads is increased (up to **16**), the execution time decreases rapidly. Beyond **16** threads, we observe smaller decreases in execution time with the smallest execution time observed when number of threads **(p) = 32**.

By increasing the number of threads, we can achieve faster execution times, as all the threads would be working on the same problem / task simultaneously. However, this is limited by the value of **k'**, i.e., the terminal matrix size which defines the upper limit for the matrix size below which the matrix multiplication would be computed serially using the traditional matrix multiplication algorithm.

Moreover, using many threads leads to increased memory usage, resource contention and context switching which eventually results in an increase in execution time and lower efficiency. No errors were observed in any of the experiments conducted.

Design Choices to improve parallel performance:

I have used OpenMP's **'#pragma omp task'** directive to explicitly define a task and identify a block of code to run in parallel with code outside the task region. It is used in parallelizing irregular algorithms, especially recursive algorithms. The calculations of **M1** to **M7** are parallelized and **'#pragma omp taskwait'** is used to wait for all the child tasks to complete, thereby providing synchronization in computing the Result matrix. In this manner, the parallel performance is improved.

Conclusions:

1. Matrix size significantly impacts the execution time of matrix multiplication. An increase in matrix size results in an increase in execution time.

2. Terminal Matrix size / Number of levels of recursion also significantly impacts the execution time of matrix multiplication. An increase in **k'** results in faster execution times as the traditional matrix multiplication algorithm will be used for smaller submatrices.

3. Number of threads utilized significantly impacts the execution time of matrix multiplication. As the number of threads are increased up to 32, the execution times decrease. Beyond 32 threads, the execution time increases due to resource contention, memory overhead and context switching.

Description to compile and execute the code:

The following commands must be run:

```
module load intel
icc -qopenmp -o major_project.exe major_project.cpp
./major_project.exe <k> <k'> <q>
```

where:
k = $\log_2$(Matrix Size)
k’ = Number of levels of recursion
q = $\log_2$(Number of threads)

Note:
Terminal matrix size / threshold = $\mathbf{2^{(k - k')}}$

Example:

```
./major_project.exe 10 8 5
```

Matrix Size = 1024
k’ = 8
Terminal Matrix Size = 4
Number of threads = 32
