PC - Major Project
Parallelizing Strassen's Matrix-Multiplication Algorithm

Gundam Satyabhama Reddy
UIN - 933004899

For this project, I have used OpenMP, as we have worked with it throughout the semester.

To Compile:
module load intel
icc -qopenmp -o matrix.exe matrix.cpp

To Run:
./matrix.exe <k> <k'> <p>

- k is the size of the initial matrices ($n = 2^k$)
- k' is the size of the matrix ($s = 2^k/2^{k'}$) where we have to terminate the recursion and use a naive algorithm to calculate the product.
- $p = \log_2$(#Threads).

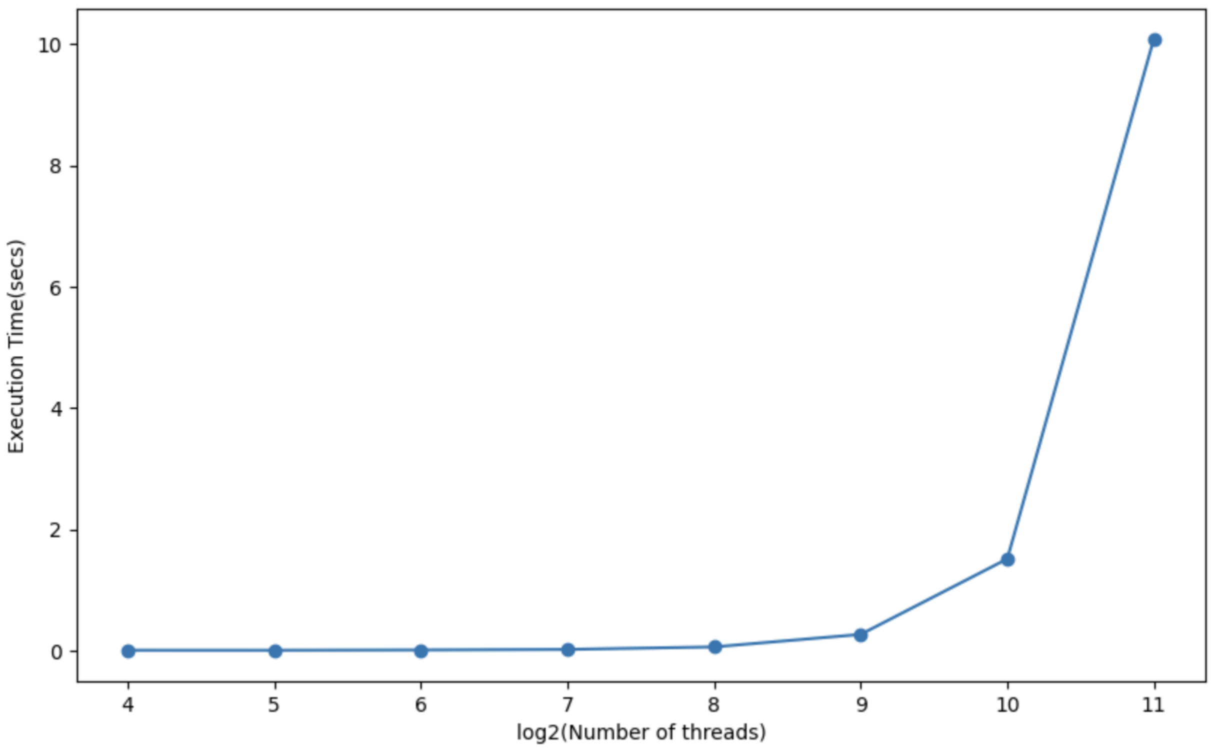
Example Execution:
./matrix.exe 4 3 4

Code Details:
My algorithm takes the above parameters as input and computes the matrix size as n*n where $n = 2^k$ and the termination matrix size as s*s where $s=(2^k/2^{k'})$. The code conducts recursive operations until the termination condition, matrix size = $(s=2^k/2^{k'})$, is met. When we reach this point, we use Strassen's Standard Matrix Multiplication, which defines how to generate matrix multiplications using M1 through M7 formulas.

Analysis

1. In the first experiment, I changed **k** from 4 to 11, which means matrices' sizes vary from 16 x 16 to 2048 x 2048. During this, k' = 3 and p = 4 (16 threads) are kept constant. Below are the logs and graph of execution times:

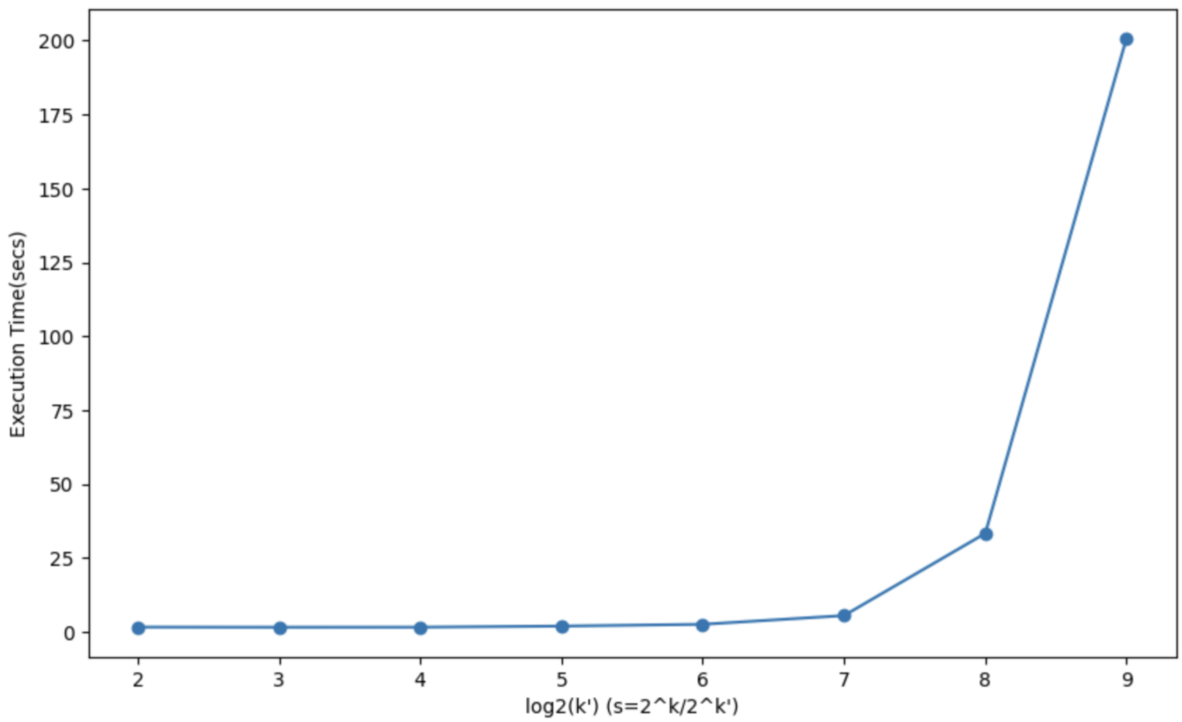
Matrix (n*n) = 16 x 16, Terminal matrix (s*s) = 2 x 2, # Threads = 16, Time = 0.004218sec
Matrix (n*n) = 32 x 32, Terminal matrix (s*s) = 4 x 4, # Threads = 16, Time = 0.003734sec
Matrix (n*n) = 64 x 64, Terminal matrix (s*s) = 8 x 8, # Threads = 16, Time = 0.008423sec
Matrix (n*n) = 128 x 128, Terminal matrix (s*s) = 16 x 16, # Threads = 16, Time = 0.017466sec
Matrix (n*n) = 256 x 256, Terminal matrix (s*s) = 32 x 32, # Threads = 16, Time = 0.058608sec
Matrix (n*n) = 512 x 512, Terminal matrix (s*s) = 64 x 64, # Threads = 16, Time = 0.264953sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 16, Time = 1.510487sec
Matrix (n*n) = 2048 x 2048, Terminal matrix (s*s) = 256 x 256, # Threads = 16, Time = 10.085076sec



We can see that as the size of the matrices increases, the execution time increases as well. The increase is drastic between 10 and 11, as the matrix size also increases by a lot.

2. In the second experiment, I changed **k’** from 2 to 9, which means the terminal matrices' sizes vary from 4 x 4 to 512 x 512. During this, k = 10 and p = 4 (16 threads) are kept constant. Below are the logs and graph of execution time:

Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 256 x 256, # Threads = 16, Time = 1.574616sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 16, Time = 1.523930sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 64 x 64, # Threads = 16, Time = 1.542931sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 32 x 32, # Threads = 16, Time = 1.910737sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 16 x 16, # Threads = 16, Time = 2.511032sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 8 x 8, # Threads = 16, Time = 5.461338sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 4 x 4, # Threads = 16, Time = 33.267927sec
Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 2 x 2, # Threads = 16, Time = 200.834147sec

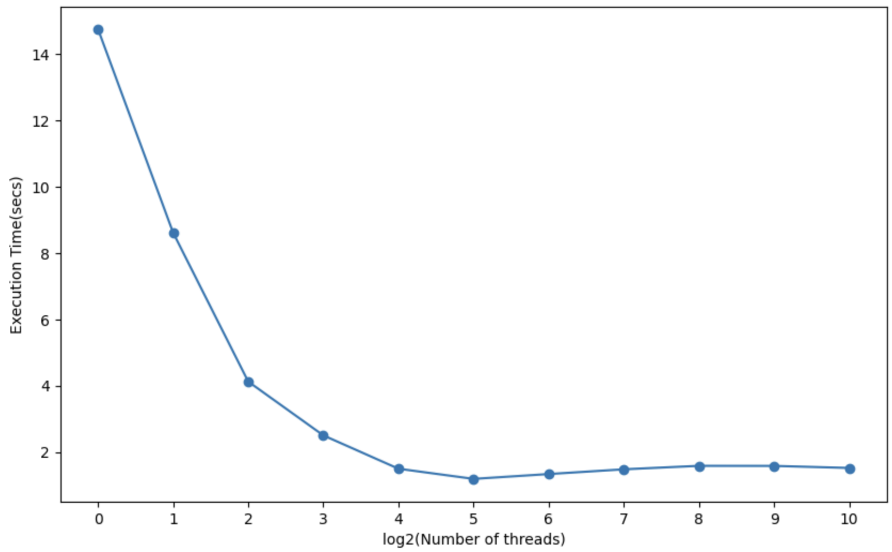


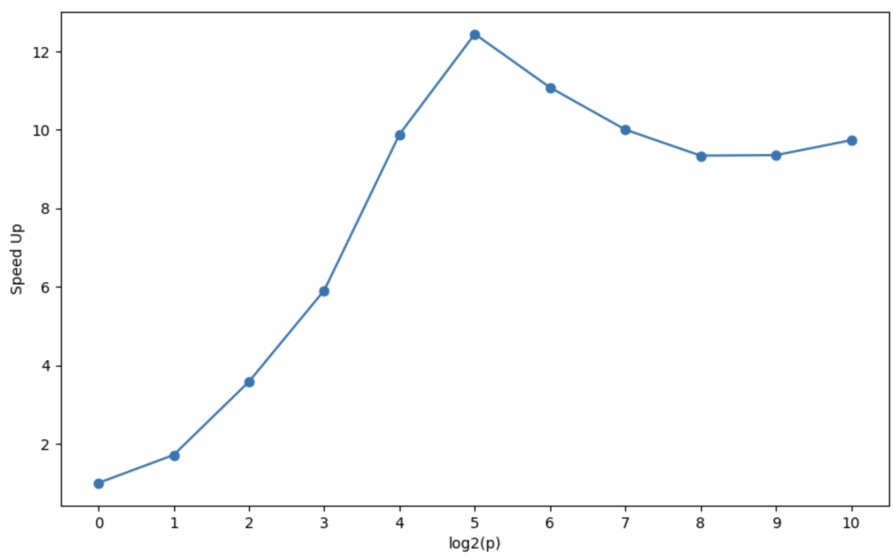
Based on the graph, we can conclude that as the value of k' increases, the size of the termination matrix reduces, and hence the time for multiplication increases. This is because a smaller termination matrix needs more recursive calls to execute and reach the termination matrix, increasing execution time. In other words, if the termination matrix size is larger, the code computes Strassen's technique earlier, resulting in a shorter overall execution time.

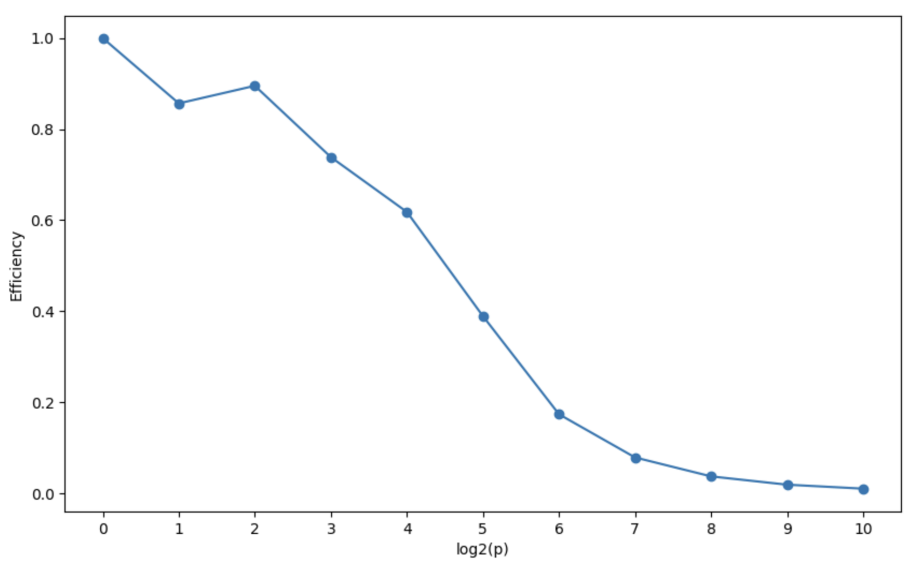
3. In the third experiment, I change the number of threads from 0 to 1024 while keeping k = 10 and k' = 3. Below are the logs and graphs of execution time, speedup, and efficiency:

   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 1, Time = 14.758219sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 2, Time = 8.616036sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 4, Time = 4.121861sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 8, Time = 2.498813sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 16, Time = 1.493233sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 32, Time = 1.186616sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 64, Time = 1.332746sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 128, Time = 1.475722sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 256, Time = 1.580352sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 512, Time = 1.578244sec
   Matrix (n*n) = 1024 x 1024, Terminal matrix (s*s) = 128 x 128, # Threads = 1024, Time = 1.516418sec

Number of Threads (p)	Execution Time (secs)	Speed Up	Efficiency
1	14.758219	1.0	1.0
2	8.616036	1.713	0.856
4	4.121861	3.58	0.895
8	2.498813	5.906	0.738
16	1.493233	9.883	0.618
32	1.186616	12.437	0.389
64	1.332746	11.074	0.173
128	1.475722	10.001	0.078
256	1.580352	9.339	0.036
512	1.578244	9.351	0.018
1024	1.516418	9.732	0.01


Execution Time(secs)
log2(Number of threads)


Speed Up
log2(p)


Efficiency
log2(p)

At a point after log(threads) = 4, there is an increase in execution time, causing speedup to decrease. This increase in time could be due to the extra time required for thread context switching as the number of threads increases. This is also evident from the efficiency graph. We can see that as the number of threads increases, the efficiency reduces due to context switching.

=================== XXXXX ===================