CSCE 735 Fall 2022

Name: Rohan Chaudhury
Email: rohan.chaudhury@tamu.edu
UIN: 432001358

Major Project:
Parallelizing Strassen's Matrix-Multiplication Algorithm

1. (75 points) In this project, you have to develop a parallel implementation of Strassen's recursive algorithm for computing matrix-multiplications. You should choose one of the following strategies.

a. Develop a shared-memory code using OpenMP, or

b. Develop a CUDA-based code for the GPU.

Your code should compute the product of two matrices of size n x n, where n=2^k, and k is an input to your program. Your code should also accept k' as an input to allow the user to vary the terminal matrix size.

Solution:

The code implementation is complete. I used OpenMP to develop my code.

The code requires three inputs:

1. The size(k),
2. the terminal size(k'), and
3. The number of threads/processes (p),

to compute the matrix multiplication of two matrices using the Strassen algorithm up to terminal size k', and then computes the remaining multiplication of matrices using the regular matrix multiplication algorithm.

Code is attached in the zip folder. I have used grace to complete the code. Following are the commands to compile and execute the code on grace portal:

```
module load intel
```

```
icc -qopenmp -o major_project.exe major_project_432001358_strassen.cpp
```

```
./major_project.exe k k' p
```

Where,

k is the matrix size

k' is the terminal matrix size

p is the number of threads

2. (25 points) Develop a report that describes the parallel performance of your code. You will have to conduct experiments to determine the execution time for different values of k, and use that data to plot speedup and efficiency graphs. You should also experiment with different values of k' to explore its impact on the execution time.

Discuss any design choices you made to improve the parallel performance of the code and how it relates to actual observations. Include any insights you obtained while working on this project.

Lastly, include a brief description of how to compile and execute the code on platform you have chosen.

Solution:

Following is the description of the parallelization process:

To compute matrix multiplication, I utilized the recursive Strassen algorithm. Because the size of the matrix is decreased to half in each recursive run of the method, so, I verified whether or not the size of the matrix is less than or equal to the terminal matrix size (k') at the beginning of the function. If the matrix size becomes less than or equal to the terminal size, I call the conventional matrix multiplication algorithm, otherwise I proceed with the recursive Strassen algorithm.

Strassen's algorithm computes the product as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix},$$

where

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}), \qquad M_2 = (A_{21} + A_{22})B_{11}, \qquad M_3 = A_{11}(B_{12} - B_{22}),$$

$$M_4 = A_{22}(B_{21} - B_{11}), \qquad M_5 = (A_{11} + A_{12})B_{22},$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}), \qquad M_7 = (A_{12} - A_{22})(B_{21} + B_{22}).$$

I have parallelized the computation of these 7 products of matrices shown above. The Strassen Multiplication function is run recursively after computation of every matrix.

The Speedup and Efficiency:

In my experiments I have used 4 different matrix sizes (k=512, 1024, 2048, 4096), varied the number of threads (p=1, 2, 4, 8, 16, 32, 64, 128) while keeping the value of terminal size k' constant (=6). The obtained data is used to determine the execution time for different values of k, and to plot speedup and efficiency graphs. Following is the experimental data obtained.

For k=512, k'=6, p=1,2,4,8,16,32,64,128

Correct Output	size n (2^k)	512	k'	6	Threads	1	Execution time (sec)	0.138395	Speedup	1	Efficiency	1
Correct Output	size n (2^k)	512	k'	6	Threads	2	Execution time (sec)	0.128031	Speedup	1.080949145	Efficiency	0.540474573
Correct Output	size n (2^k)	512	k'	6	Threads	4	Execution time (sec)	0.07986	Speedup	1.732970198	Efficiency	0.433242549
Correct Output	size n (2^k)	512	k'	6	Threads	8	Execution time (sec)	0.054461	Speedup	2.541176255	Efficiency	0.317647032
Correct Output	size n (2^k)	512	k'	6	Threads	16	Execution time (sec)	0.038934	Speedup	3.554605229	Efficiency	0.222162827
Correct Output	size n (2^k)	512	k'	6	Threads	32	Execution time (sec)	0.040208	Speedup	3.441976721	Efficiency	0.107561773
Correct Output	size n (2^k)	512	k'	6	Threads	64	Execution time (sec)	0.207992	Speedup	0.665386169	Efficiency	0.010396659
Correct Output	size n (2^k)	512	k'	6	Threads	128	Execution time (sec)	0.407011	Speedup	0.340027665	Efficiency	0.002656466

For k=1024, k’=6, p=1,2,4,8,16,32,64,128

Correct Output	size n (2^k)	1024	k'	6	Threads	1	Execution time (sec)	0.565146	Speedup	1	Efficiency	1
Correct Output	size n (2^k)	1024	k'	6	Threads	2	Execution time (sec)	0.343752	Speedup	1.644051526	Efficiency	0.822025763
Correct Output	size n (2^k)	1024	k'	6	Threads	4	Execution time (sec)	0.188699	Speedup	2.994960228	Efficiency	0.748740057
Correct Output	size n (2^k)	1024	k'	6	Threads	8	Execution time (sec)	0.113701	Speedup	4.970457604	Efficiency	0.6213072
Correct Output	size n (2^k)	1024	k'	6	Threads	16	Execution time (sec)	0.079217	Speedup	7.134150498	Efficiency	0.445884406
Correct Output	size n (2^k)	1024	k'	6	Threads	32	Execution time (sec)	0.088695	Speedup	6.371790969	Efficiency	0.199118468
Correct Output	size n (2^k)	1024	k'	6	Threads	64	Execution time (sec)	0.206728	Speedup	2.733766108	Efficiency	0.042715095
Correct Output	size n (2^k)	1024	k'	6	Threads	128	Execution time (sec)	1.460885	Speedup	0.386851806	Efficiency	0.00302228

For k=2048, k’=6, p=1,2,4,8,16,32,64,128

Correct Output	size n (2^k)	2048	k'	6	Threads	1	Execution time (sec)	3.518911	Speedup	1	Efficiency	1
Correct Output	size n (2^k)	2048	k'	6	Threads	2	Execution time (sec)	1.863337	Speedup	1.888499504	Efficiency	0.944249752
Correct Output	size n (2^k)	2048	k'	6	Threads	4	Execution time (sec)	0.997754	Speedup	3.526832265	Efficiency	0.881708066
Correct Output	size n (2^k)	2048	k'	6	Threads	8	Execution time (sec)	0.558186	Speedup	6.304190718	Efficiency	0.78802384
Correct Output	size n (2^k)	2048	k'	6	Threads	16	Execution time (sec)	0.349657	Speedup	10.06389404	Efficiency	0.628993378
Correct Output	size n (2^k)	2048	k'	6	Threads	32	Execution time (sec)	0.323053	Speedup	10.89267396	Efficiency	0.340396061
Correct Output	size n (2^k)	2048	k'	6	Threads	64	Execution time (sec)	0.461256	Speedup	7.6289761	Efficiency	0.119202752
Correct Output	size n (2^k)	2048	k'	6	Threads	128	Execution time (sec)	0.940371	Speedup	3.742045427	Efficiency	0.02923473

For k=4096, k’=6, p=1,2,4,8,16,32,64,128

Correct Output	size n (2^k)	4096	k'	6	Threads	1	Execution time (sec)	28.050261	Speedup	1	Efficiency	1
Correct Output	size n (2^k)	4096	k'	6	Threads	2	Execution time (sec)	14.227599	Speedup	1.971538627	Efficiency	0.985769314
Correct Output	size n (2^k)	4096	k'	6	Threads	4	Execution time (sec)	7.34373	Speedup	3.819620411	Efficiency	0.954905103
Correct Output	size n (2^k)	4096	k'	6	Threads	8	Execution time (sec)	3.830511	Speedup	7.32285092	Efficiency	0.915356365
Correct Output	size n (2^k)	4096	k'	6	Threads	16	Execution time (sec)	2.14498	Speedup	13.07716669	Efficiency	0.817322918
Correct Output	size n (2^k)	4096	k'	6	Threads	32	Execution time (sec)	1.37078	Speedup	20.4629926	Efficiency	0.639468519
Correct Output	size n (2^k)	4096	k'	6	Threads	64	Execution time (sec)	1.626836	Speedup	17.24221802	Efficiency	0.269409657
Correct Output	size n (2^k)	4096	k'	6	Threads	128	Execution time (sec)	2.861691	Speedup	9.801988055	Efficiency	0.076578032

And following are the **speedup and efficiency graphs** for the four different matrix sizes:

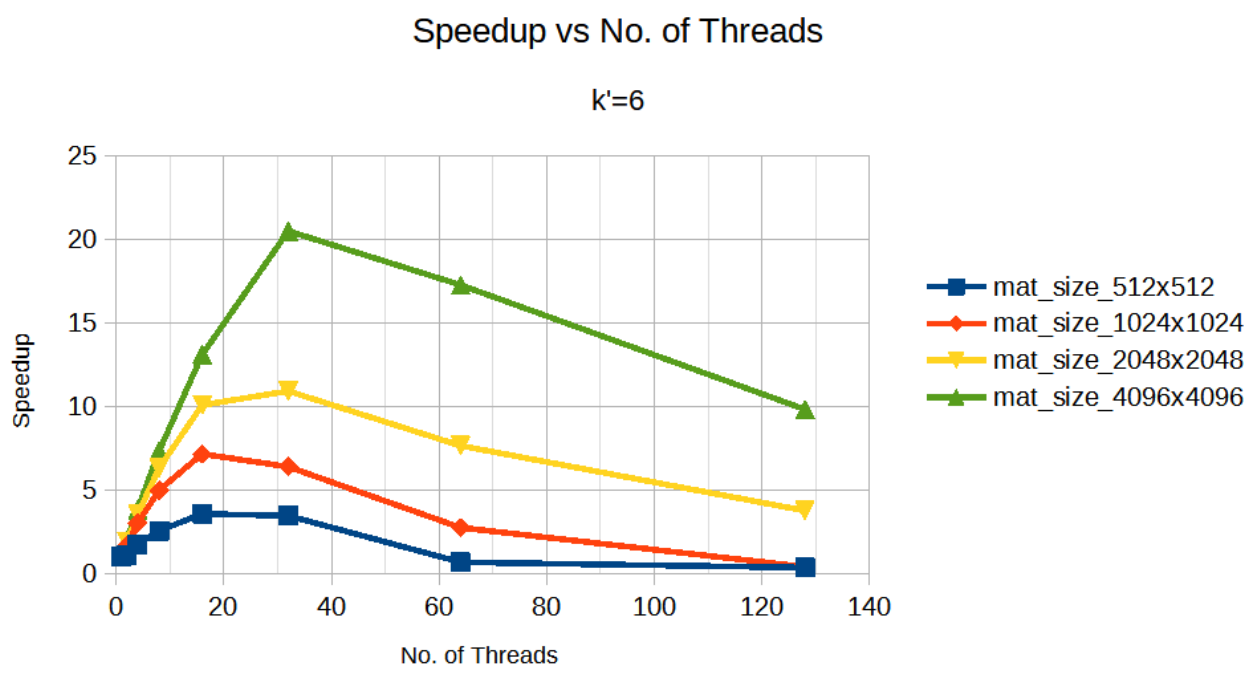


Fig 1.

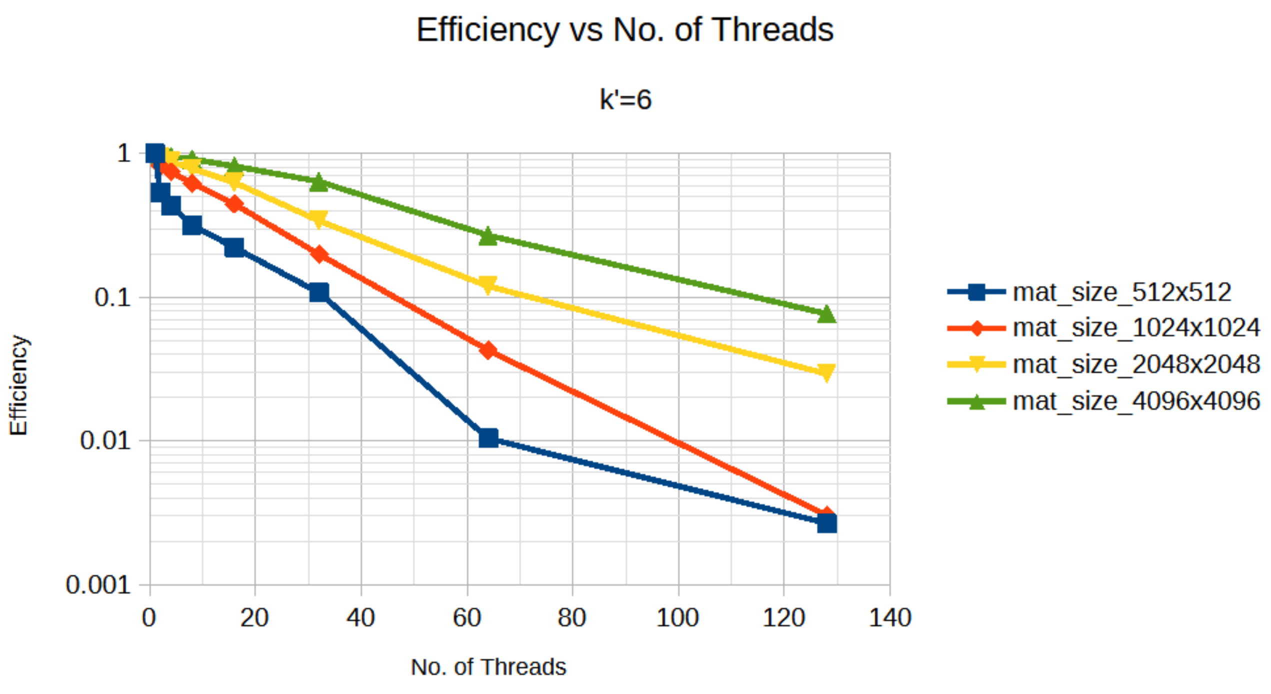


Fig 2.

Observation:

In the graphs shown above, the blue, red, yellow, and green lines in Fig 1. represent Speed up versus Thread count and in Fig 2. represents the Efficiency vs Thread Count across different values of Matrix Size.

We can see from Fig 1 that as the matrix size increases the associated **speed up** also increases. This is to be expected since, when the input size/data to be worked on is minimal, having multiple threads would increase execution time owing to higher thread management overhead costs when compared to single threaded execution. In this case, the thread management overhead is nearly equal to the time necessary to execute the operation in a single threaded environment.

However, when the size of the matrix grows, single threaded execution takes significantly longer. As a result, the thread management overhead durations in parallelization implementations will not cause **parallelized execution times** to exceed single threaded execution times in this case.

Similar behavior can be seen in Fig 1, where raising the number of threads improves speed up until an optimal value is achieved, after which speed begins to deteriorate due to the aforementioned thread management overheads.

From Fig 1, we can also observe that as the matrix size increases, the optimum speed up is achieved at greater thread counts. This may be explained using the ideas of work division. The number of work units that a thread can process has an ideal value at some point, after which it may require multiple context switches or significant memory accesses to complete its tasks. Longer execution times arise from this circumstance. Additionally, if the number of work units that must be processed by the threads falls below this ideal work unit load for the threads, the CPU will spend more time on thread handling rather than carrying out their intended tasks. We may expect increased thread counts to result in optimal performance for bigger matrix sizes based on this.

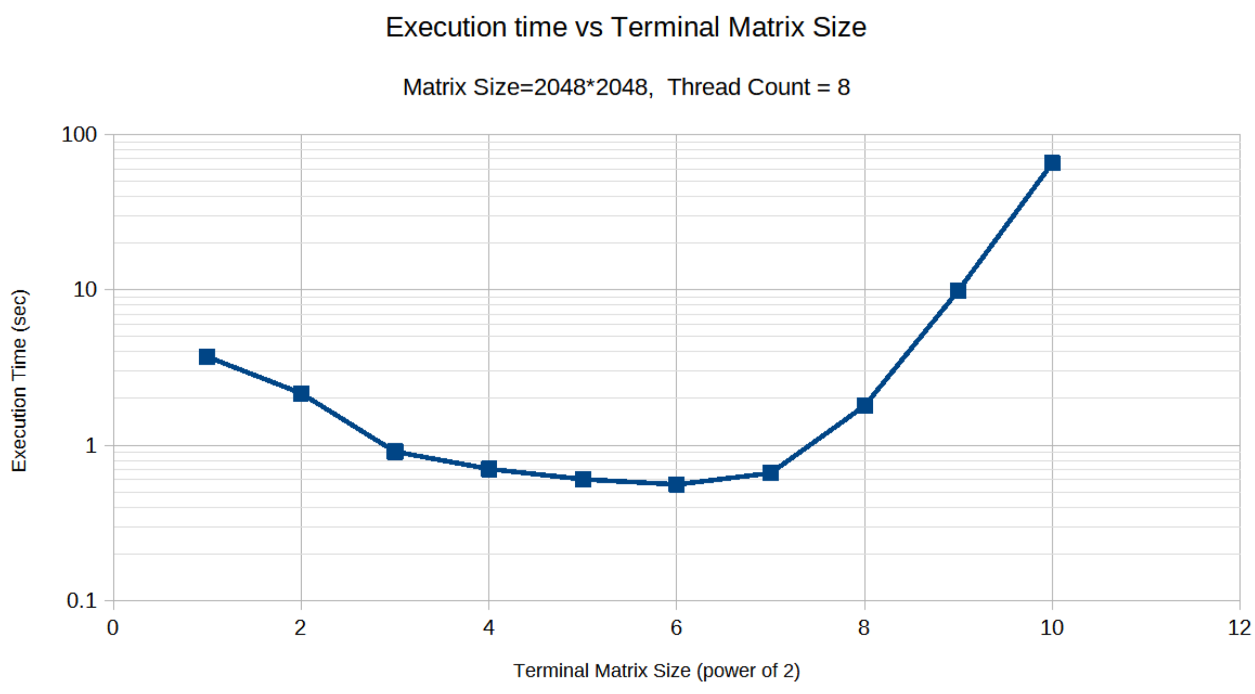
From the efficiency vs. number of threads plot in Fig 2, we can see that the efficiency declines as the number of threads increases. This is caused because of thread synchronization and thread contention over computational tasks.

Execution time vs k':

Varying execution times were achieved for different k' (k'=1 to 10), while keeping k=11 (n = 2^k = 2048) and no. of threads, p=8 constant throughout.

It was noted that k' = 6 provided the shortest execution time as is evident from the table and graph below.

Correct Output	size n (2^k)	2048	k'	1	Threads	8	Execution time (sec)	3.712481
Correct Output	size n (2^k)	2048	k'	2	Threads	8	Execution time (sec)	2.161174
Correct Output	size n (2^k)	2048	k'	3	Threads	8	Execution time (sec)	0.908313
Correct Output	size n (2^k)	2048	k'	4	Threads	8	Execution time (sec)	0.704059
Correct Output	size n (2^k)	2048	k'	5	Threads	8	Execution time (sec)	0.603464
Correct Output	size n (2^k)	2048	k'	6	Threads	8	Execution time (sec)	0.559029
Correct Output	size n (2^k)	2048	k'	7	Threads	8	Execution time (sec)	0.661146
Correct Output	size n (2^k)	2048	k'	8	Threads	8	Execution time (sec)	1.791224
Correct Output	size n (2^k)	2048	k'	9	Threads	8	Execution time (sec)	9.910744
Correct Output	size n (2^k)	2048	k'	10	Threads	8	Execution time (sec)	65.593748



Following are a few observations that I made:

1. As we raise k', conventional matrix multiplication is called later, that is, on smaller matrix sizes, and therefore runtime decreases since the majority of the computation is done via the parallel Strassen algorithm.

2. As k' grows big, a significant number of recursive calls aggregate due to a large number of calls to the recursive Strassen algorithm, resulting in a longer runtime.
3. At k=6' we get the optimum conditions for the algorithm to work resulting in a least execution time.

Design Choices:

1. The function accepts as inputs k, k', and the number of threads and outputs the time it takes to implement Strassen's matrix multiplication method.
2. The matrix dimensions are n*n, where n = 2^k.
3. The size of the terminal matrix is calculated as s*s, where s = $(2^k)/(2^{k'})$.
4. The code computes the matrix multiplication of two matrices using the Strassen algorithm up to terminal size k', and then computes the remaining multiplication of matrices using the regular matrix multiplication algorithm.
5. Each of the seven matrices described in the Strassen Algorithm is enveloped in an OpenMP task to provide a synchronized and parallel computation.

Compilation and Execution:

Code is attached in the zip folder. I have used grace to develop and test the code. The experiments were conducted using batch files. Following are the commands to compile and execute the code on grace portal:

module load intel

icc -qopenmp -o major_project.exe major_project_432001358_strassen.cpp

./major_project.exe k k' p

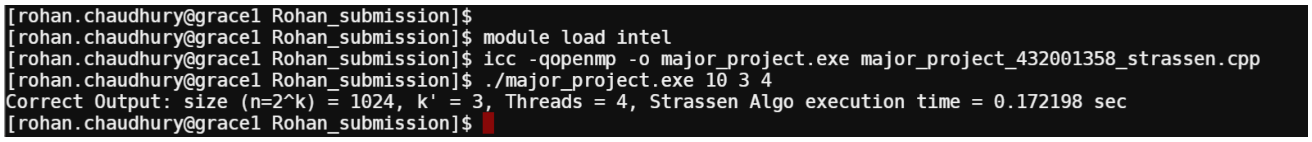
Where,

k is the matrix size

k' is the terminal matrix size

p is the number of threads+

Example Execution snapshot:



```
[rohan.chaudhury@grace1 Rohan_submission]$
[rohan.chaudhury@grace1 Rohan_submission]$ module load intel
[rohan.chaudhury@grace1 Rohan_submission]$ icc -qopenmp -o major_project.exe major_project_432001358_strassen.cpp
[rohan.chaudhury@grace1 Rohan_submission]$ ./major_project.exe 10 3 4
Correct Output: size (n=2^k) = 1024, k' = 3, Threads = 4, Strassen Algo execution time = 0.172198 sec
[rohan.chaudhury@grace1 Rohan_submission]$
```