CSCE 735 Spring 2023

Parallel Computing - Major Project

Name: Rajendra Thottempudi **UIN : 933004901**

Major Project: Parallelizing Strassen's Matrix-Multiplication Algorithm

1. 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.

2. 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.

Solution :

I developed a shared-memory based code using OpenMp for the implementation of Strassen's recursive algorithm for computing matrix multiplications.

The algorithm for matrix multiplication proposed by Volker Strassen is better than the regular algorithm because it has a lower time complexity for large matrices. The regular algorithm for matrix multiplication has a time complexity of O(n^3), where n is the size of the matrix. This means that as the size of the matrix increases, the time taken to multiply the matrices increases exponentially.

On the other hand, Strassen's algorithm has a time complexity of O(n^log2(7)), which is approximately O(n^2.81). This is better than the regular algorithm for large matrices because the exponent is smaller, which means that the time taken to multiply the matrices increases at a slower rate as the size of the matrix increases.

Strassen's algorithm achieves this improved time complexity by recursively dividing the matrices into smaller submatrices and performing matrix operations on these submatrices. It uses a clever technique to reduce the number of multiplications required by the regular algorithm, which leads to the improved time complexity. However, Strassen's algorithm has a higher constant factor than the regular algorithm, which means that for small matrices, the regular algorithm may be faster. Therefore, Strassen's algorithm is typically used for large matrices where the improved time complexity outweighs the higher constant factor.

I have tried with a list size of 1024, k' as 7 and with the processes count varying from 1 to 1024. It can be observed that Increasing the number of threads can lead to a decrease in execution time because more threads can perform the same operation in parallel, resulting in reduced workload and faster processing. The parameter k' specifies the threshold size of the array from which serial multiplication is used instead of parallel. In the case of increasing the number of threads from 1 to 32, the execution time decreased as more processes were involved in the computation. However, this trend is not

always sustained as further increase in the number of threads can lead to increased context switching, cache misses and interprocess communication, eventually leading to an increase in execution time.

As the matrix size becomes smaller and the number of threads increases, the efficiency also decreases. This is due to the allocation of threads to work, which leads to some threads being in idle state, and the overhead of starting and terminating the thread pool, which outweighs productive activity. Moreover, limited hardware resources, such as shared memory, add to the issue. The efficiency in such circumstances can decrease significantly. Threads entail various locking methods, context switching, and data transferring between threads, all of which result in additional overhead. When many threads share limited hardware resources and have the same priority, switching between them until execution is complete also adds to the overhead. Therefore, while the execution time decreases as the degree of parallelism increases and the duration drops, it increases when many threads are involved, as previously mentioned.

Design Choices :

To enhance parallel performance, we opt for using OpenMP to parallelize loops, which enables multiple threads to access the loops simultaneously. We also employ certain design choices to ensure synchronization between threads. For instance, we use "#pragma omp taskwait" to specify a wait for child tasks that are generated by the current task, thereby synchronizing the resultant matrix. In addition, the values from M1 to M7 are calculated parallelly ensuring synchronization.

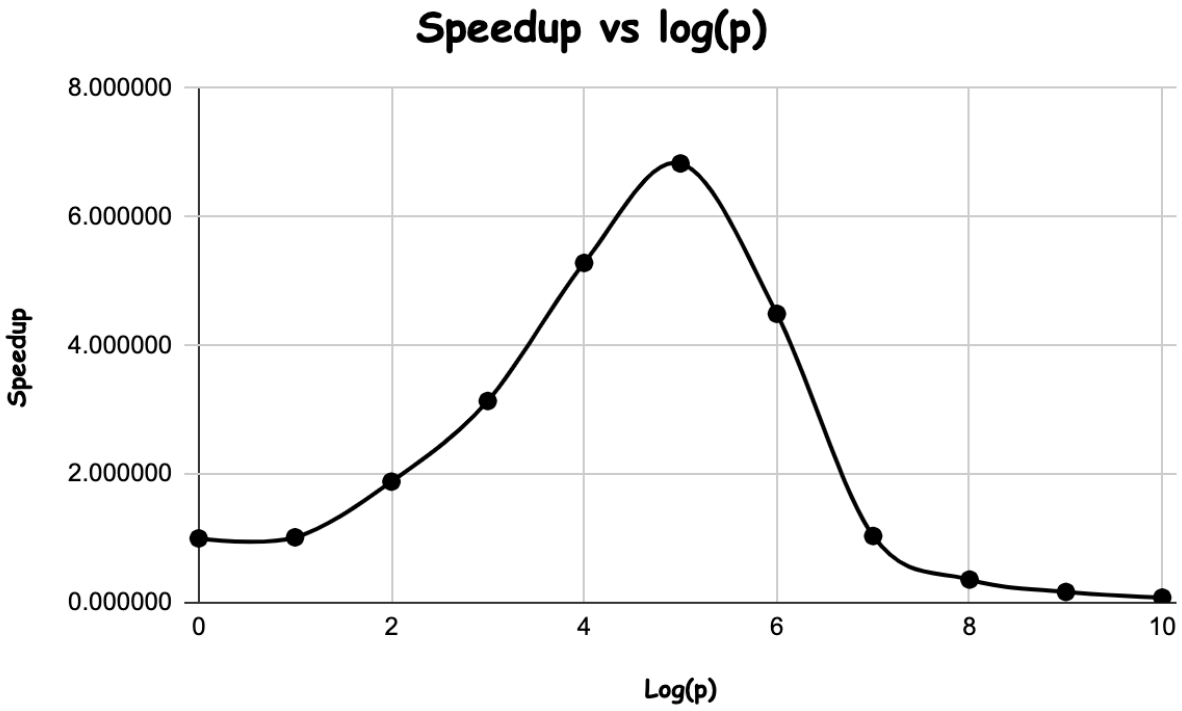
The results obtained are as follows :

Process	Execution Time	Matrix Size(k)	Threshold(k-k')	log(p)	k'
1	0.884517	1024	3	0	7
2	0.870032	1024	3	1	7
4	0.469822	1024	3	2	7
8	0.281996	1024	3	3	7
16	0.167411	1024	3	4	7
32	0.129489	1024	3	5	7
64	0.196892	1024	3	6	7
128	0.853001	1024	3	7	7
256	2.460274	1024	3	7	7
512	5.297194	1024	3	7	7
1024	11.176983	1024	3	7	7

log(p)	Speedup
0	1.000000
1	1.016649
2	1.882664
3	3.136630
4	5.283506
5	6.830827
6	4.492397
7	1.036947
8	0.359520
9	0.166978
10	0.079137

log(p)	Efficiency
0	1.000000
1	0.508324
2	0.470666
3	0.392079
4	0.330219
5	0.213463
6	0.070194
7	0.008101
8	0.001404
9	0.000326
10	0.000077

Plots obtained are as follows :


Speedup vs log(p)
8.000000
6.000000
4.000000
2.000000
0.000000
Speedup
0
2
4
6
8
10
Log(p)


Efficiency vs Log(p)
1.000000
0.750000
0.500000
0.250000
0.000000
Efficiency
0
2
4
6
8
10
Log(p)

The following data has been obtained after certain experimentation and the results obtained for various values of k' keeping k as constant have been observed as follows :

Process	Execution Time	Matrix Size(k)	k	k'
64	0.17516	1024	10	3
64	0.167703	1024	10	4
64	0.207784	1024	10	5
64	0.238204	1024	10	6
64	0.172663	1024	10	7
64	0.311499	1024	10	8
64	1.827736	1024	10	9
1	0.753042	1024	10	3
1	0.57189	1024	10	4
1	0.5125	1024	10	5
1	0.551048	1024	10	6
1	0.916821	1024	10	7
1	2.724187	1024	10	8
1	14.685269	1024	10	9

For Process = 64 :

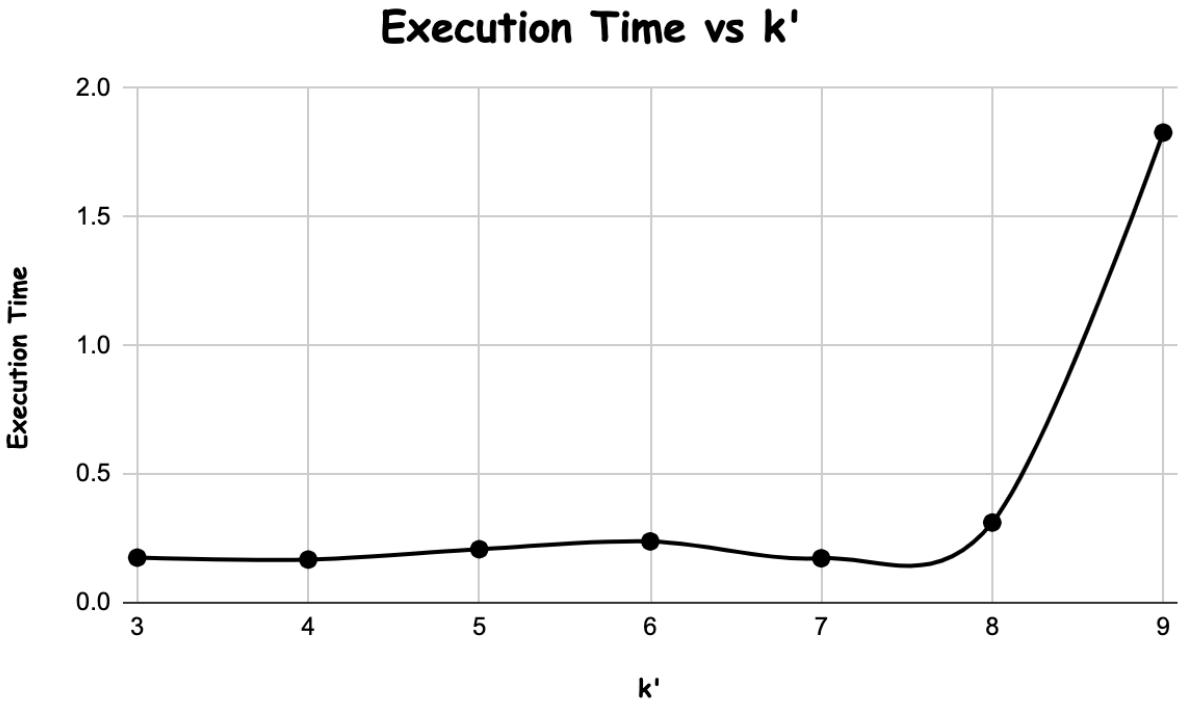
k'	Execution Time
3	0.17516
4	0.167703
5	0.207784
6	0.238204
7	0.172663
8	0.311499
9	1.827736

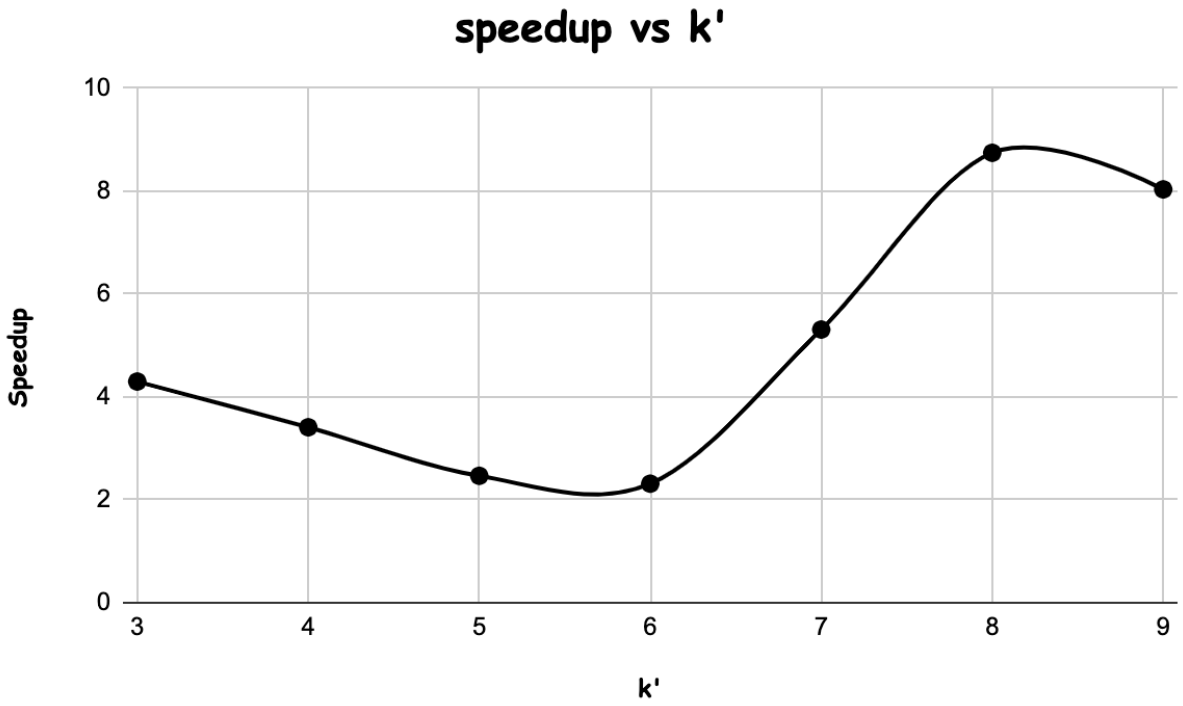
k'	Speedup

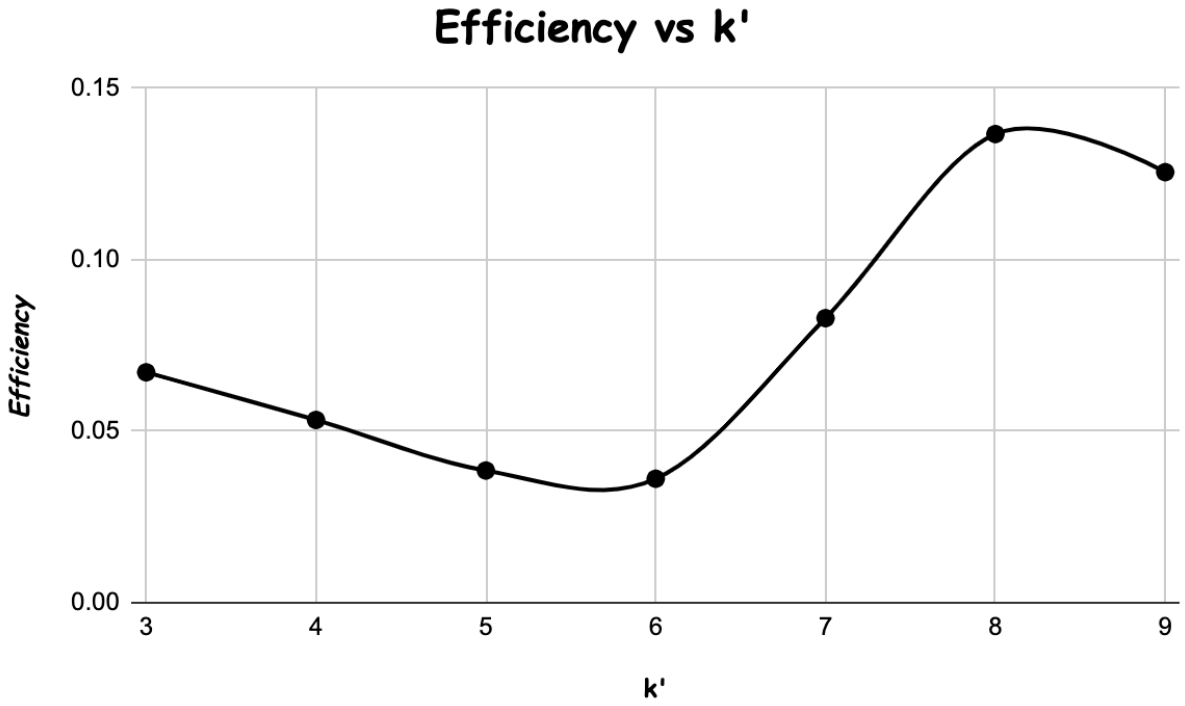
3	4.299166476
4	3.410135776
5	2.466503677
6	2.313344864
7	5.309886889
8	8.745411703
9	8.034677328

k'	Efficiency
3	0.06717447619
4	0.0532833715
5	0.03853911995
6	0.0361460135
7	0.08296698265
8	0.1366470579
9	0.1255418332

Plots are obtained as follows :


Execution Time vs k'
Execution Time
2.0
1.5
1.0
0.5
0.0
3
4
5
6
7
8
9
k'


speedup vs k'
Speedup
10
8
6
4
2
0
3
4
5
6
7
8
9
k'



It can be generally observed that when we have to multiply large matrices using serial multiplication, it takes longer to execute. However, adding more processes than needed is not helpful because it wastes resources. Having too many processes may lead to some processes sitting idle or doing nothing productive. Alternatively, the tasks may be divided into smaller parts, and the threads handling them may require more time for communication, memory reads, cache misses, scheduling overheads, and process management. Thus, it is very important to select k and k' in a manner such that optimum results are obtained using the current design of the algorithm.

Following are the compilation steps to execute :

1. `module load intel`
2. `icc -qopenmp -o major_project.exe major_project.cpp`
3. Different combinations of k, k' and threads could not be executed on the compiled file as follows :
   ./major_project.exe 10 9 64

   Where k = 10, k' = 9 and threads = 64.

Similarly, any combinations could be tried and executed.