Parallelizing Strassen's Matrix Multiplication

Sanket Hanagandi- 201IT154
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
sankethanagandi.201it154@nitk.edu.in

Mayur Jinde - 201IT135
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
jindemayur.201it135@nitk.edu.in

Shaulendra Kumar - 201IT159
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
shaulendragmailcom.201it159@nitk.edu.in

Abstract—**Matrix Multiplication is possibly one of the most important matrix operations and is actively employed in multiple scientific and engineering applications. But the main problem faced during computing the product of matrices is that matrix multiplication is a computationally intensive algorithm. The worst time complexity of the traditional or naive Matrix Multiplication algorithm is $O(n^3)$ due to which this algorithm is not suitable for larger sizes of matrices. Strassen's Matrix Multiplication Algorithm is a newer and interesting approach to compute the product of matrices and manages to reduce the worst time complexity. Strassen's Matrix Multiplication Algorithm uses a Divide and Conquer approach to bring down the time complexity to approximately $O(n^{2.81})$. But this approach has its own shortcomings as this algorithm is potentially more memory intensive and has a larger constant factor which makes this algorithm unsuitable for multiplication of smaller matrices. This paper focuses on improving the execution time of matrix multiplication by using standard parallel computing practices to perform parallel matrix multiplication. OpenMP, MPI and CUDA are used to develop algorithms by combining the naive matrix multiplication algorithm and Strassen's matrix multiplication algorithm to create hybrid algorithms which allow parallel computation of matrix multiplication to give better results in the form of smaller execution time. The algorithms are written in C++ programming language and results produced by each of the parallel hybrid algorithms are analysed to decide which algorithm works the best.**

Keywords— Strassen's Algorithm, OpenMP, MPI, CUDA, matrix multiplication

I. Introduction

Over the years, Parallel Computing has become one of the most important areas of research in the history of Computer Science. As the problems faced in the modern era become more and more complex and computationally intensive, the use of parallel computation to solve such complex problems is also increasing day by day. Parallel Computing is the process of decomposing a large or computationally intensive problems into multiple smaller, independent and similar sub-problems that can be executed simultaneously using multiple execution units, communicating using shared memory. The results of each execution unit is then combined to reconstruct the result of the final problem. Simultaneous execution reduces the total time required to compute the solution of complex problems and thus, saves time and resources. Recent improvements in hardware and software as well as the development of CPUs and GPUs has led to massive advances in the field of parallel computing. This has also led to increased use of parallel computing techniques to solve various computationally intensive problems like Matrix multiplication.

Matrix Multiplication is a very common and important operation and finds applications in almost all engineering fields. Some of the areas in which matrix multiplication is widely used include Network theory, Linear Algebra, Coordinate systems, Simulations, Population Modelling, Machine Learning etc. The Naive matrix multiplication algorithm takes $O(n^3)$ to compute the product of matrices which makes it a computationally very intensive task. Strassen's algorithm gives an improvement over the Naive matrix multiplication algorithm by slightly reducing the number of sub problems which brings down the time complexity to around $O(n^{2.81})$. But this algorithm consumes a lot of space and also has a larger constant factor which makes it unsuitable for computing product of smaller matrices. So, parallel computing can be used to improve the execution time for matrix multiplication and a hybrid algorithm can be developed which allows faster multiplication of matrices of different sizes.

This paper involves the integration of parallel computing techniques to perform matrix multiplication. Different parallelising practices such as OpenMP, MPI and CUDA are employed to develop parallel algorithms which use a combination of both the naive and strassen's matrix multiplication algorithms to compute the product of matrices of different sizes in a faster way. OpenMP uses the concept of multi threading to perform parallelism in uniprocessor system. MPI or Message Passing Interface enables communication between multiple execution cores to perform parallel execution whereas CUDA uses GPUs to perform parallel execution. The execution time of these algorithms is used to create plots and compare the algorithms. Finally, the results are displayed at the end and the conclusions derived from the results are mentioned.

II. Related Work

[1] In this paper, the authors have implemented serial and parallel traditional and Strassen's matrix multiplication. OpenMP has been used for parallelizing Strassen's algorithm. The sequential and parallel programs were ran on Intel Pentium and Intel T4500 dual core processors. The programs were ran on different matrix dimensions and multiple trials were performed to infer the conclusion. The results showed that Strassen's Algorithm gave best results when one level recursion was used. However, only OpenMP was used to parallelize Strassen's multiplication and the algorithm was not compared with MPI or CUDA.

[2] This paper focuses mainly on accelerating high performance applications using CUDA and MPI. Strassen's matrix multiplication has been used for the analysis purpose. CUDA+MPI both were used in a single program for parallelizing Strassen's matrix multiplication. The program was run under two different environments based on computer specifications. The results here showed that MPI based implementation gave better results than the CUDA+MPI based implementation. It was due to the lack of second level parallelism in Strassen's algorithm.

[3] The authors of this paper developed efficient GPU implementations of Strassen's and Winograd's matrix multiplication algorithm. NVIDIA C1060 GPU was used to execute these algorithms. Each of these algorithms were tested on different levels. For instance, 1-level, 2-level, etc implementations were used for analysing the speedup obtained in Strassen's algorithm. The speedups achieved by Strassen's and Winograd's algorithms were 32% and 33% respectively on the given GPU specification.

III. Problem Statement

Parallelizing Strassen's matrix multiplication using OpenMP, MPI and CUDA and analyzing its behaviour against the traditional matrix multiplication algorithms.

IV. Methodology

This section contains the description of all the algorithms that are used for performing matrix multiplication. First, the sequential matrix multiplication algorithms are mentioned which include the conventional *Naive* and *Strassen's* algorithms which perform matrix multiplication using sequential execution. Next up, the parallel hybrid matrix multiplication algorithms are mentioned. These algorithms are created by combining the naive and strassen's algorithms. Further, parallel execution is made possible by using OpenMP, MPI or CUDA.

A. Sequential Algorithm

Let X and Y be 2 matrices of dimension $N \times N$, the naive way to perform matrix multiplication contains 3 nested loops each of N iterations. Hence the naive matrix multiplication algorithm is computationally expensive as it has an asymptotic upper bound of $O(N^3)$.

Algorithm 1 Sequential Naive Matrix Multiplication

```
procedure SequentialNaive(X, Y)
    $N \leftarrow X.dim$
    $Z \leftarrow Matrix(N \times N)$
    for $i \leftarrow 1$ to $N$ do
        for $j \leftarrow 1$ to $N$ do
            for $k \leftarrow 1$ to $N$ do
                $Z_{ij} \leftarrow Z_{ij} + X_{ik} \times Y_{kj}$
            end for
        end for
    end for
    return $Z$
end procedure
```

Strassen's matrix multiplication is a divide and conquer-based approach. It divides the problem into 7 smaller sub-problems and solves them recursively. Each sub-problem involves multiplying 2 matrices of dimension $\frac{N}{2} \times \frac{N}{2}$. The time complexity can be represented as

$$T(N) = 7 \times T(\frac{N}{2}) + O(N^2) \quad (1)$$

Using master theorem with *a* = 7, *b* = 2 and *f(N)* = $O(N^2)$, the time complexity can be simplified as

$$T(N) = O(N^{\log 7}) = O(N^{2.8074}) \quad (2)$$

Let X and Y be 2 matrices of dimension $N \times N$. Strassen's divide and conquer approach divides these matrices into 4 quadrants each. Let A, B, C and D be the four quadrants of matrix X. Similarly let E, F, G and H be the four quadrants of matrix Y, each of these will have a dimension of $\frac{N}{2} \times \frac{N}{2}$. Then the product matrix Z can be represented using eq. (3)

$$Z = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} \quad (3)$$

These 8 submatrices are used to build 7 smaller sub-problems as shown below.

$$S_1 = (B - D) \times (G + H) \quad (4)$$

$$S_2 = (A + D) \times (E + H) \quad (5)$$

$$S_3 = (A - C) \times (E + F) \quad (6)$$

$$S_4 = (A + B) \times H \quad (7)$$

$$S_5 = A \times (F - H) \quad (8)$$

$$S_6 = D \times (G - E) \quad (9)$$

$$S_7 = (C + D) \times E \quad (10)$$

These sub-problems are recursively solved, and their solutions are combined to construct the four components of the final product matrix, namely P, Q, R and S. The sub-matrices P, Q, R and S are calculated using eq. (11), eq. (12), eq. (13) and eq. (14) respectively.

$$P = S_1 + S_2 - S_4 + S_6 \tag{11}$$

$$Q = S_4 + S_5 \tag{12}$$

$$R = S_6 + S_7 \tag{13}$$

$$S = S_2 - S_3 + S_5 - S_7 \tag{14}$$

The components are combined together to form the product matrix Z using eq. (15)

$$Z = \begin{bmatrix} P & Q \\ R & S \end{bmatrix} \tag{15}$$

Strassen's algorithm shows poor performance as compared to the Naive algorithm for smaller matrices. The below pseudocode shows a hybrid Strassen's algorithm, which switches to the naive algorithm when the size of the matrix is smaller than a certain cutoff.

Algorithm 2 Sequential Strassen's Matrix Multiplication

```
procedure SequentialStrassen(X, Y)
    if $X.dim \leq CUTOFF$ then
        return $SequentialNaive(X, Y)$
    end if
    $A, B, C, D \leftarrow Decompose(X)$
    $E, F, G, H \leftarrow Decompose(Y)$
    $S_1 \leftarrow SequentialStrassen(B - D, G + H)$
    $S_2 \leftarrow SequentialStrassen(A + D, E + H)$
    $S_3 \leftarrow SequentialStrassen(A - C, E + F)$
    $S_4 \leftarrow SequentialStrassen(A - B, H)$
    $S_5 \leftarrow SequentialStrassen(A, F - H)$
    $S_6 \leftarrow SequentialStrassen(D, G - E)$
    $S_7 \leftarrow SequentialStrassen(C + D, E)$
    $P \leftarrow S_1 + S_2 - S_4 + S_6$
    $Q \leftarrow S_4 + S_5$
    $R \leftarrow S_6 + S_7$
    $S \leftarrow S_2 - S_3 + S_5 - S_7$
    $Z \leftarrow Combine(P, Q, R, S)$
    return $Z$
end procedure
```

B. Parallel Algorithm

Parallel algorithms for matrix multiplication are developed by combining Naive and Strassen's matrix multiplication algorithms. Parallel execution of these hybrid algorithms is done by using application programming interfaces such as OpenMP, MPI, or CUDA.

1) OpenMP: OpenMP (Open Multi-Processing) is an application programming interface that is explicitly used to perform multi-threaded, shared memory parallelism. The proposed algorithm uses the concept of *OpenMP tasks* and *collapsing of for loops* to enable parallel execution of matrix multiplication using OpenMP. An OpenMP task is created for each of the 7 recursive calls in Strassen's algorithm as shown in Algorithm 3, which enables the computation of matrices S_1, S_2,..., S_7 parallelly leading to better performance.

Algorithm 3 Parallel Strassen's Algorithm (OpenMP)

```
procedure ParallelNaive(X, Y)
    $N \leftarrow X.dim$
    $Z \leftarrow Matrix(N \times N)$
    parallel omp for collapse(2)
        for $i \leftarrow 1$ to $N$ do
            for $j \leftarrow 1$ to $N$ do
                for $k \leftarrow 1$ to $N$ do
                    $Z_{ij} \leftarrow Z_{ij} + X_{ik} \times Y_{kj}$
                end for
            end for
        end for
    end parallel
    return $Z$
end procedure

procedure ParallelStrassenOmp(X, Y)
    if $X.dim \leq CUTOFF$ then
        return $ParallelNaive(X, Y)$
    end if
    $A, B, C, D \leftarrow Decompose(X)$
    $E, F, G, H \leftarrow Decompose(Y)$
    create omp task
        $S_1 \leftarrow ParallelStrassenOmp(B - D, G + H)$
    create omp task
        $S_2 \leftarrow ParallelStrassenOmp(A + D, E + H)$
    create omp task
        $S_3 \leftarrow ParallelStrassenOmp(A - C, E + F)$
    create omp task
        $S_4 \leftarrow ParallelStrassenOmp(A - B, H)$
    create omp task
        $S_5 \leftarrow ParallelStrassenOmp(A, F - H)$
    create omp task
        $S_6 \leftarrow ParallelStrassenOmp(D, G - E)$
    create omp task
        $S_7 \leftarrow ParallelStrassenOmp(C + D, E)$
    omp taskwait

    create omp task
        $P \leftarrow S_1 + S_2 - S_4 + S_6$
    create omp task
        $Q \leftarrow S_4 + S_5$
    create omp task
        $R \leftarrow S_6 + S_7$
    create omp task
        $S \leftarrow S_2 - S_3 + S_5 - S_7$
    omp taskwait

    $Z \leftarrow Combine(P, Q, R, S)$
    return $Z$
end procedure
```

When a thread encounters a task, it may choose to execute the task or place the task in a conceptual pool of tasks. Other threads can then take tasks from the pool and execute them until the pool is empty. After all recursive calls are completed, tasks are created again for computing the sub-matrices P, Q, R, and S to further parallelize the algorithm. As Strassen's algorithm doesn't work well for smaller matrices thus, when the size of matrices is smaller than a certain cutoff, multiplication of smaller matrices is done using the naive algorithm. To enable parallel execution of the naive algorithm, the *for* loops are collapsed using OpenMP, and chunks of iterations are distributed among the various threads. Further, care has been taken to free the matrices as soon as they are not needed anymore so that algorithm remains memory efficient.

2) MPI: It is a standardized system that allows exchange of messages between multiple computers running a program across distributed memory. The same program is executed by multiple processes leading to parallelism.

Algorithm 4 Parallel Strassen's Algorithm (MPI)

```
procedure ParallelStrassenMpi(X, Y)
    $A, B, C, D \leftarrow Decompose(X)$
    $E, F, G, H \leftarrow Decompose(Y)$
    if rank = 1 then
        $S_1 \leftarrow SequentialStrassen(B - D, G + H)$
        $MPI\_Send(S_1 \rightarrow rank_0)$
    else if rank = 2 then
        $S_2 \leftarrow SequentialStrassen(A + D, E + H)$
        $MPI\_Send(S_2 \rightarrow rank_0)$
    else if rank = 3 then
        $S_3 \leftarrow SequentialStrassen(A - C, E + F)$
        $MPI\_Send(S_3 \rightarrow rank_0)$
    else if rank = 4 then
        $S_4 \leftarrow SequentialStrassen(A - B, H)$
        $MPI\_Send(S_4 \rightarrow rank_0)$
    else if rank = 5 then
        $S_5 \leftarrow SequentialStrassen(A, F - H)$
        $MPI\_Send(S_5 \rightarrow rank_0)$
    else if rank = 6 then
        $S_6 \leftarrow SequentialStrassen(D, G - E)$
        $MPI\_Send(S_6 \rightarrow rank_0)$
    else if rank = 7 then
        $S_7 \leftarrow SequentialStrassen(C + D, E)$
        $MPI\_Send(S_7 \rightarrow rank_0)$
    else if rank = 0 then
        $MPI\_Recv(S_1, S_2, S_3, S_4, S_5, S_6, S_7)$
    end if
    $MPI\_Barrier()$

    if rank = 0 then
        $P \leftarrow S_1 + S_2 - S_4 + S_6$
        $Q \leftarrow S_4 + S_5$
        $R \leftarrow S_6 + S_7$
        $S \leftarrow S_2 - S_3 + S_5 - S_7$
        $Z \leftarrow Combine(P, Q, R, S)$
    end if
    return $Z$
end procedure
```

Each process performs computations on its local variables and then communicates with other processes through message passing to get the final results. To perform parallel execution of matrix multiplication, the MPI environment is initialized with 8 processes. Each process has a rank assigned to it, starting from rank 0 to rank 7. The initial 7 sub-problems or recursive calls are assigned to processes with rank 1 to rank 7. So, the process with rank 1 executes the 1st recursive call, the process with rank 2 executes the 2nd recursive call, and so on. Each process then uses the hybrid sequential Strassen's algorithm (Algorithm 2) to perform further multiplication of matrices. Each process then uses an MPI routine to send its results to the process with rank 0 and the process with rank 0, in turn, receives the resultant matrices of each of the other processes. Finally, the process with rank 0 computes the sub-matrices P, Q, R, and S and then combines the sub-matrices to get the final result. Algorithm 4 displays the pseudo-code for the matrix multiplication algorithm, which uses MPI for parallel execution.

C. CUDA

CUDA is an Application Programming Interface (API) that harnesses the power of Graphical Processing Units (GPUs) to perform parallelism. It is a parallel computing platform developed by NVIDIA for performing parallel computation on GPUs. GPUs are processing units that are specialized to perform tasks that require massive parallelisms such as high-resolution image/ video rendering, machine learning, and various scientific computation. GPUs have thousands of low capacity cores and ALUs (Arithmetic Logical Units) and are capable of performing a huge number of repetitive low-level tasks simultaneously. They specialize in data processing rather than flow control and data caching.

As CUDA specializes in performing computationally intensive, highly parallel tasks thus, it can be used to perform parallel computation of Matrix multiplication. For performing matrix multiplication using CUDA, both naive and Strassen's methods are used. The parallel Naive CUDA matrix multiplication algorithm uses two functions, one to be executed by the host and the other to be executed by each thread of GPU. First, the matrices are copied from the host memory to the device memory. Then, the CUDA kernel is launched by specifying the dimensions of the grids and the blocks of threads which are to be used for matrix multiplication. As a larger number of threads in a block is more efficient thus, the number of threads per block is kept as large as possible. All mentioned threads execute the device function where the id of each thread is calculated on the basis of the position of the block in the grid, the dimension of the block as well as the position of the thread in the block. Based on the thread id, each thread is tasked to calculate the value of one cell of the product matrix, and the row and column of this cell are decided using the thread id. For Strassen's matrix multiplication using CUDA, the main problem is divided into 7 sub-problems by the host CPU, and each of the sub-problems is computed parallelly using CUDA.

Algorithm 5 Parallel Naive Algorithm (CUDA)

```
deviceProcedure MULTIPLY($X_d, Y_d, Z_d$)
    $N \leftarrow X_d.dim$
    $id \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
    $i \leftarrow id \ / \ N$
    $j \leftarrow id \ \% \ N$
    for $k \leftarrow 0$ to $N - 1$ do
        $Z_{dij} \leftarrow Z_{dij} + X_{dik} \times Y_{dkj}$
    end for
end deviceProcedure

hostProcedure PARALLELNAIVECUDA($X_h, Y_h$)
    $N \leftarrow X_h.dim$
    $bytes = sizeof(X_h)$
    $cudaMalloc(X_d, bytes)$
    $cudaMalloc(Y_d, bytes)$
    $cudaMalloc(Z_d, bytes)$
    $cudaMemcpy(X_d \leftarrow X_h)$
    $cudaMemcpy(Y_d \leftarrow Y_h)$

    $threads \leftarrow min(max\_threads, N)$
    $blocks \leftarrow N^2/threads$
    $gridDims \leftarrow [\ blocks \times 1 \times 1\ ]$
    $blockDims \leftarrow [\ threads \times 1 \times 1\ ]$
    $multiply < gridDims, blockDims > (X_d, Y_d, Z_d)$
    $cudaDeviceSynchronize()$

    $cudaMemcpy(Z_h \leftarrow Z_d)$
    $cudaFree(X_d)$
    $cudaFree(Y_d)$
    $cudaFree(Z_d)$
    return $Z_h$
end hostProcedure
```

Algorithm 6 Parallel Strassen's Algorithm (CUDA)

```
procedure PARALLELSTRASSENCUDA(X, Y)
    if $X.dim \leq CUTOFF$ then
        return $SequentialNaive(X, Y)$
    end if
    $A, B, C, D \leftarrow Decompose(X)$
    $E, F, G, H \leftarrow Decompose(Y)$
    $S_1 \leftarrow ParallelNaiveCuda(B - D, G + H)$
    $S_2 \leftarrow ParallelNaiveCuda(A + D, E + H)$
    $S_3 \leftarrow ParallelNaiveCuda(A - C, E + F)$
    $S_4 \leftarrow ParallelNaiveCuda(A - B, H)$
    $S_5 \leftarrow ParallelNaiveCuda(A, F - H)$
    $S_6 \leftarrow ParallelNaiveCuda(D, G - E)$
    $S_7 \leftarrow ParallelNaiveCuda(C + D, E)$
    $P \leftarrow S_1 + S_2 - S_4 + S_6$
    $Q \leftarrow S_4 + S_5$
    $R \leftarrow S_6 + S_7$
    $S \leftarrow S_2 - S_3 + S_5 - S_7$
    $Z \leftarrow Combine(P, Q, R, S)$
    return $Z$
end procedure
```

V. Results and Analysis

Table 1 shows the execution time when each of the algorithms (Algorithm 1 to Algorithm 6) are executed with different sizes of matrices. Algorithms 1 and 2 are sequential algorithms, whereas Algorithms 3, 4, 5 and 6 use parallel computing. In algorithms that require a cutoff, the value is set to 32. This means the algorithm switches to naive algorithm when the size of matrices involved in multiplication is less than or equal to 32. For Algorithm 5 and 6 which uses CUDA, *Tesla K80* is used as GPU. The sequential, OpenMP and MPI programs are run on Intel i5 8th Gen processor.

- As the size of matrices increase, the execution times of each of the algorithms also increase. This is obvious as the product of larger matrices will require more time.
- Further, it can be observed that sequential algorithms perform well for smaller sizes of matrices. Almost in all cases, the hybrid sequential Strassen's algorithm performs better than the naive algorithm except for CUDA implementations.
- As the size of matrices increase, the performance of sequential algorithms decreases rapidly. Comparatively, the parallel hybrid algorithms show much better performance even for products of matrices with larger sizes.
- For OpenMP implementation, it can be observed that for larger sizes of matrices, the implementation which uses 8 threads performs better than the implementations which use 2 or 4 threads. This is because more number of threads allows faster execution of the created tasks.
- For CUDA implementation, it can be observed that the naive CUDA implementation performs better than Strassen's CUDA implementation. This is because GPUs perform best in scenarios where a large number of low-capacity cores perform the same operation on different data. Strassen's matrix multiplication algorithm uses recursion and converts the main problem into 7 subproblems. Launching the GPU kernel for each of the subproblems involves a considerable overhead, which in turn adds some more time to the overall runtime. Thus, the naive CUDA matrix multiplication algorithm works faster.
- For a matrix of size 512, Strassen's MPI implementation gives a speedup of 3x, and OpenMP implementation with 8 threads gives a speedup of 3.5x, CUDA implementation gives 4x and Naive CUDA the best performance with a speedup of 133x as compared to the Strassen sequential algorithm.
- When the matrix size is 1024, Parallel Strassen(MPI) gives a speedup of 3x, Parallel Strassen(OpenMP) with 8 threads, Parallel Starssen(CUDA), and Parallel Naive(CUDA) implementations gives speedup of 4x and 16x, 98x respectively.
- Furthermore, when the matrix size is 2048, Parallel

TABLE I: Runtime(in sec) at different matrix sizes

Algo/Dims	128	256	512	1024	2048
Naive Seq	0.0133	0.1103	1.0837	15.424	190.92
Strassen Seq	0.0120	0.0763	0.5211	3.6056	25.430
Strassen OpenMP (2)	0.0081	0.0342	0.2282	1.5360	12.536
Strassen MPI	0.0034	0.0497	0.2000	1.2937	8.3417
Strassen OpenMP (4)	0.0037	0.0255	0.1447	0.8754	5.8095
Strassen OpenMP (8)	0.0228	0.0586	0.1488	0.8174	5.2808
Strassen CUDA	0.1212	0.1246	0.1428	0.2260	0.6580
Naive CUDA	0.0002	0.0007	0.0039	0.0366	0.2834

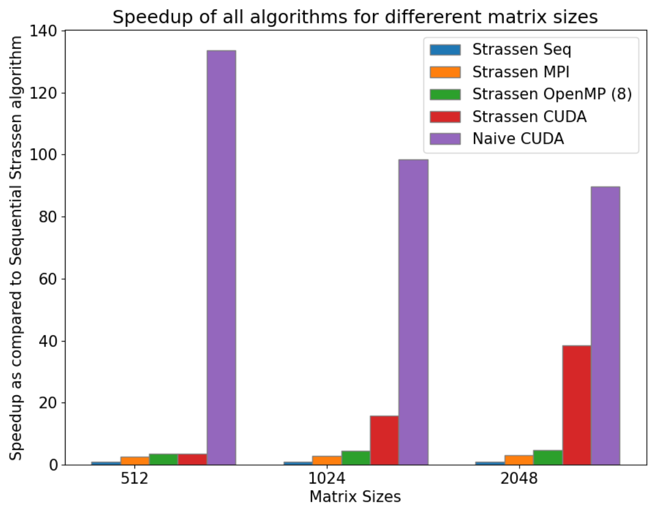


Fig. 1: Bar graph of speedups of algorithms with respect to Sequential Strassen

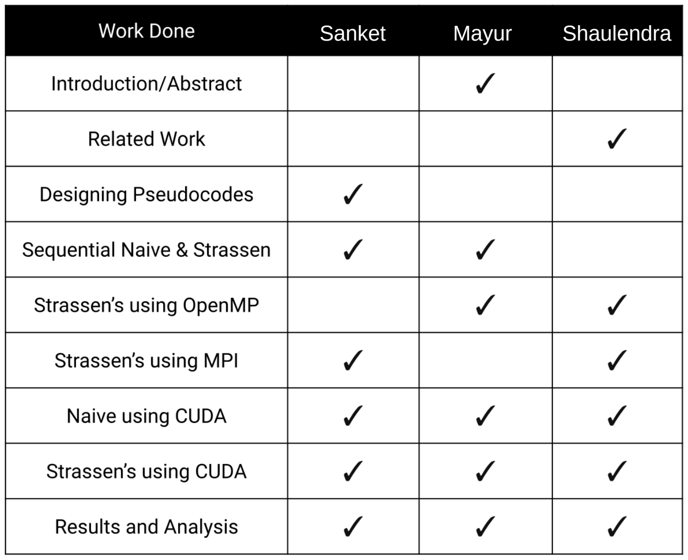
Strassen(MPI) gives a speedup of 3x, whereas the OpenMP with 8 threads, Strassen's CUDA and Naive CUDA implementations gives speedup of 5x, 39x, and 90x respectively.

- This shows that parallelism using CUDA gives the best results, followed by OpenMP and then MPI. Sequential Strassen comes next, and Sequential Naive algorithm performs the worst. The parallel algorithms dominate over the sequential algorithms, especially when the size of matrices is large.
- Among parallel algorithms, CUDA performs the best and outperforms all other algorithms by large margins. This is because CUDA uses a large number of lightweight threads to perform matrix multiplication parallelly. But the downside of this algorithm requires additional hardware, i.e., GPUs, which are quite expensive.
- Further, among parallel algorithms, OpenMP performs better than MPI in most cases. This is mainly because OpenMP allows the creation of tasks and dependencies between tasks that are more suited for executing recursive calls. Further, OpenMP threads can work using shared memory, whereas each process in MPI requires its own copy of data to perform matrix multiplication.

VI. Conclusion

Matrix multiplication is a computationally intensive operation with a time complexity of $O(n^3)$. Strassen's algorithm provides an improvement over the naive algorithm by bringing the time complexity down to $O(n^{2.81})$, but even this improvement is not enough. This paper aims to reduce the time required to perform matrix multiplication by using parallel computing. Hybrid algorithms are proposed which use a combination of both Naive and Strassen's algorithm and then introduce parallelism by using OpenMP, MPI, or CUDA. As is evident from the results, the proposed parallel hybrid algorithms show better performance as compared to the sequential algorithms. Further, it is observed that CUDA performs the best for this purpose, but, the algorithm can only be used when GPUs are available which can be very expensive. After CUDA, OpenMP performs the best and is closely followed by MPI. This shows that parallel computation using OpenMP, MPI or CUDA can massively improve the execution time of computationally intensive tasks like matrix multipication.

Individual Contribution



Work Done	Sanket	Mayur	Shaulendra
Introduction/Abstract		✓	
Related Work			✓
Designing Pseudocodes	✓		
Sequential Naive & Strassen	✓	✓	
Strassen's using OpenMP		✓	✓
Strassen's using MPI	✓		✓
Naive using CUDA	✓	✓	✓
Strassen's using CUDA	✓	✓	✓
Results and Analysis	✓	✓	✓

Fig. 2: Individual Contribution

References

[1] A. J. Kawu, A. Yahaya Umar and S. I. Bala, ”Performance of one-level recursion parallel Strassen’s algorithm on dual core processor,” 2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON), 2017, pp. 587-591, doi: 10.1109/NIGERCON.2017.8281929.

[2] N. P. Karunadasa and D. N. Ranasinghe, ”Accelerating high performance applications with CUDA and MPI,” 2009 International Conference on Industrial and Information Systems (ICIIS), 2009, pp. 331-336, doi: 10.1109/ICIINFS.2009.5429842.

[3] J. Li, S. Ranka and S. Sahni, ”Strassen’s Matrix Multiplication on GPUs,” 2011 IEEE 17th International Conference on Parallel and Distributed Systems, 2011, pp. 157-164, doi: 10.1109/ICPADS.2011.130.

[4] P. Chen, K. Dai, D. Wu, J. Rao and X. Zou, ”The parallel algorithm implementation of matrix multiplication based on ESCA,” 2010 IEEE Asia Pacific Conference on Circuits and Systems, 2010, pp. 1091-1094, doi: 10.1109/APCCAS.2010.5774970.

[5] P. Rathod, A. Vartak and N. Kunte, ”Optimizing the complexity of matrix multiplication algorithm,” 2017 International Conference on Intelligent Computing and Control (I2C2), 2017, pp. 1-4, doi: 10.1109/I2C2.2017.8321772.

[6] M. I. Soliman and F. S. Ahmed, ”Exploiting ILP, DLP, TLP, and MPI to accelerate matrix multiplication on Xeon processors,” 2014 International Conference on Engineering and Technology (ICET), 2014, pp. 1-6, doi: 10.1109/ICEngTechnol.2014.7016779.

[7] Matrouk, Khaled and Alhasanat, Abdullah and Alashaary, Haitham and Alqadi, Ziad and AL-Shalabi, Hasan, ”Analysis of Matrix Multiplication Computational Methods,” 2014 European Journal of Scientific Research. 121. 258-266.