Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

# Parallel Processing: Assignment 2

## Objective:

The well-known matrix multiplication (MM) problem is fundamental to many high-performance applications. The goal of this assignment is to compute $|X^k|$ in parallel, where

• X is an n X n matrix

• k is an input parameter

To solve the above problem, three different parallel formulations were used. These include the traditional $n^3$ algorithm with data decomposition, Cannons' method, and Strassen's method. Below are the details for each of the formulations along with the MPI calls used.

Commonly used MPI Calls between the three formulations:

| **MPI Calls** | **Description** |
|---|---|
| int MPI_Init(int *argc, char ***argv) | used to Initialize the MPI execution environment |
| Int MPI_Comm_size(MPI_Comm comm, int *size) | Determines the size of the group associated with a communicator |
| int MPI_Comm_rank(MPI_Comm comm, int *rank) | Determines the rank of the calling process in the communicator |
| int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart) | Makes a new communicator to which Cartesian topology information has been attached. |
| int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]) | Determines process coordinates in cartesian topology given rank in group |
| double MPI_Wtime() | Returns an elapsed time on the calling processor. |
| int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) | Reduces values on all processes within a group. |
| int MPI_Comm_free(MPI_Comm *comm) | Marks the communicator object for deallocation |
| int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int | Sends personalized data from one process to all other processes in a communicator |

| | |
|---|---|
| recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm) | |
| int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) | Performs a blocking send |
| int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status) | Blocking receive for a message |
| int MPI_Barrier(MPI_Comm comm) | Blocks until all processes in the communicator have reached this routine. |
| int MPI_Finalize() | Terminates MPI execution environment |

**Conventional n$^3$ matrix multiplication algorithm:**

The algorithm for the conventional matrix multiplication is given below.

```
procedure MAT_MULT (A, B, C)
begin
   for i := 0 to n - 1 do
       for j := 0 to n - 1 do
           begin
              C[i, j] := 0;
              for k := 0 to n - 1 do
                  C[i, j] := C[i, j] + A[i, k] x B[k, j];
           endfor;
end MAT_MULT
```

In our parallel formulation of the above algorithm,

- Initially, we create a 2D mesh topology of the processors where each row and column of the mesh consists of root p processors. Where p is the total number of processors
- We then use the block-block data decomposition technique to assign a subset of the total matrix A to each of the processors.
- Each processor is assigned (n/p) rows and (n/p) columns of data from the complete matrix. Here n is the number of rows/columns in the initial square matrix.
- The resultant matrix C is the output of the local product matrices on each of the processors.
- The final matrix is calculated by performing a gather on the root node of all the resultant sub-matrices.

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

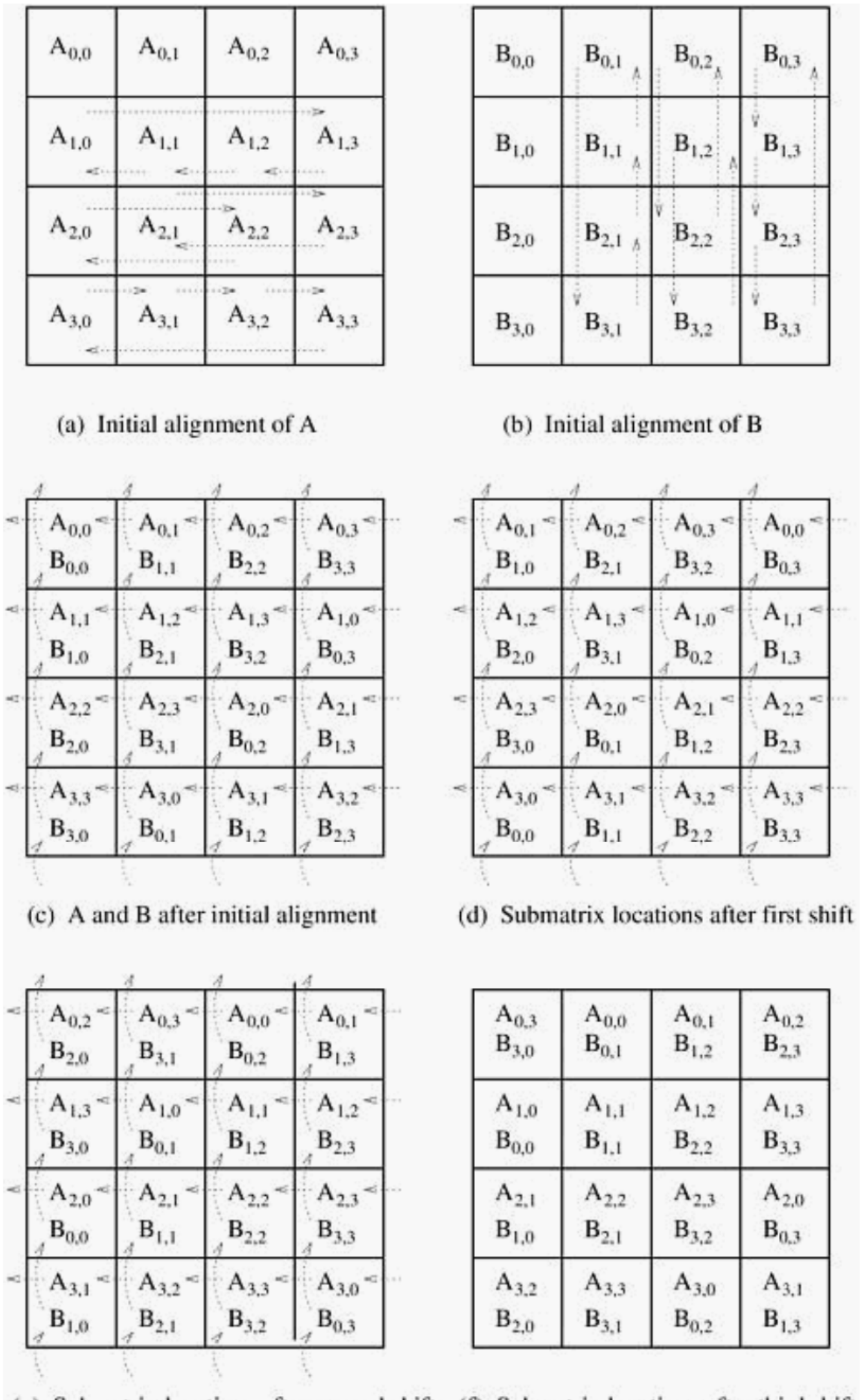
**Cannon's Matrix Multiplication:**

This is a more memory-efficient version of the algorithm presented in the previous formulation. This algorithm works best on computers where the processors are connected in an n x n mesh. In our current formulation,

- The initial matrix X is partitioned into p square blocks
- The processors are labeled from $P_{0,0}$ to $P_{(root(p-1),\ root(p-1)).}$ The submatrices $X_{i,j}$ is assigned to processor $P_{i,j}$. This is the checkerboard pattern of data decomposition.
- To compute the matrix multiplication, all processors in the ith row would need all submatrices containing it. This is achieved by rotating the submatrix $A_{i,k}$ among the processes.
- A similar approach is used for all the columns as well and the matrix multiplication of each of the submatrices can be calculated.
- The above steps are repeated root p times to get the final result of the matrix multiplication.

The advantage of cannon's algorithm is that the total memory requirement over all the processors is $\theta(n^2)$ which is independent of the number of processors and only dependent on the size of the input matrix.

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283



(a) Initial alignment of A (b) Initial alignment of B

(c) A and B after initial alignment (d) Submatrix locations after first shift

(e) Submatrix locations after second shift (f) Submatrix locations after third shift

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

The overall parallel run time of the algorithm is:

$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}.$$

And its isoefficiency function is $\Theta(p^{3/2})$.

The MPI Calls used to achieve this are:

| **MPI Calls** | **Description** |
|---|---|
| int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest) | Returns the shifted source and destination ranks, given a shift direction and amount |
| int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status * status) | Sends and receives using a single buffer |

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

**Strassen's matrix multiplication:**

Strassen's matrix multiplication algorithm is a recursive approach to matrix multiplication. In each recursive step, the matrix is divided into four submatrices of size n/2 X n/2. In order to better understand the algorithm, let us consider an example. Let A, and B be two matrices of size 2 X 2, and let C be the resultant matrix. They can be represented as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

The intermediate step of Strassen's algorithm defines a new matrix given by the formulas:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}); \\ M_2 &= (A_{21} + A_{22})B_{11}; \\ M_3 &= A_{11}(B_{12} - B_{22}); \\ M_4 &= A_{22}(B_{21} - B_{11}); \\ M_5 &= (A_{11} + A_{12})B_{22}; \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}); \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}), \end{aligned}$$

The final resulting matrix is given by:

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

The parallel implementation of the algorithm was done as follows:

- A mesh of size root p x root p is created among the p processors.
- The initial matrix X is divided into 4 sub matrices of size n/2 X n/2.
- The processor that did the division of the matrix keeps one of the four parts and then send the remaining three parts to 3 other processors in the mesh.
- The intermediate values are now calculated on all the processors and the result is sent back to the root process
- The final matrix values are calculated at the root to give us the result of the matrix multiplication.

The overall time complexity of Strassen's algorithm is:

$$T(N) = 7T(N/2) + O(N^2)$$

The iso efficiency function is:

The MPI calls used to implement this algorithm are:

| **MPI Calls** | **Description** |
|---|---|
| int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest) | Returns the shifted source and destination ranks, given a shift direction and amount |
| int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status * status) | Sends and receives using a single buffer |
| int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request) | Begins a nonblocking receive |
| int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) | Begins a nonblocking send |
| int MPI_Wait(MPI_Request *request, MPI_Status *status) | Waits for an MPI request to complete |
| int MPI_Type_commit(MPI_Datatype *datatype) | Commits the datatype |
| MPI_Type_create_subarray()int MPI_Type_create_subarray(int ndims, const int array_of_sizes[], const int array_of_subsizes[], const int array_of_starts[], int order, MPI_Datatype oldtype,MPI_Datatype *newtype) | Create a datatype for a subarray of a regular, multidimensional array |
| int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype) | Create a datatype with a new lower bound and extent from an existing datatype |

| | |
|---|---|
| int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm ) | Broadcasts a message from the process with rank "root" to all other processes of the communicator |
| int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) | Sends data from one process to all other processes in a communicator |
| int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) | Gathers together values from a group of processes |

Although the algorithm provides better time complexity compared to the traditional matrix multiplication algorithm, it is not space-efficient and is memory intensive. This is due to the fact that the sub-matrices recursion process takes extra space

**Parameter Ranges:**

To test the run times for the above formulations, we have considered the following ranges for the 3 parameters of matrix size, power k, and the number of processors.

Matrix size: From 64 x 64 to 256 x 256

Power k: 2 to 4

Number of processors: 4 to 64

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

**Results**:

| Matrix Size | K | Number of Processes | Serial Time (Ts) | Parallelized traditional Matrix Multiplication in Mesh | Speedup | Efficiency |
|---|---|---|---|---|---|---|
| 32 * 32 | 4 | 16 | 0.000279 | 0.008701 | 0.032059 | 0.002004 |
| 64 * 64 | 2 | 4 | 0.001828 | 0.001237 | 1.523608 | 0.380902 |
| 64 * 64 | 2 | 16 | 0.002015 | 0.007324 | 0.278264 | 0.017392 |
| 128 * 128 | 2 | 4 | 0.015468 | 0.003823 | 4.046024 | 1.011506 |
| 128 * 128 | 2 | 16 | 0.017109 | 0.002863 | 5.932967 | 0.370810 |
| 256 * 256 | 4 | 64 | 0.162713 | 0.012935 | 12.579360 | 0.196552 |

| Matrix Size | K | Number of Processes | Serial Time (Ts) | Cannon's Algorithm | Speedup | Efficiency |
|---|---|---|---|---|---|---|
| 32 * 32 | 4 | 16 | 0.000279 | 0.002698 | 0.103393 | 0.006462 |
| 64 * 64 | 2 | 8 | 0.001828 | 0.000724 | 2.525362 | 0.631341 |
| 64 * 64 | 2 | 16 | 0.002015 | 0.008417 | 0.239385 | 0.014962 |
| 128 * 128 | 2 | 4 | 0.015468 | 0.002236 | 6.917369 | 1.729342 |
| 128 * 128 | 2 | 16 | 0.017109 | 0.009526 | 1.783231 | 0.111452 |
| 256 * 256 | 4 | 64 | 0.162713 | 0.013358 | 12.412901 | 0.193952 |

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

| Matrix Size | K | Number of Processes | Serial Time (Ts) | Strasson's Algorithm | Speedup | Efficiency |
|---|---|---|---|---|---|---|
| 32 * 32 | 4 | 16 | 0.000279 | 0.005311 | 0.052523 | 0.003283 |
| 64 * 64 | 2 | 8 | 0.001828 | 0.007094 | 0.252337 | 0.063084 |
| 64 * 64 | 2 | 16 | 0.002015 | 0.004260 | 0.472968 | 0.029561 |
| 128 * 128 | 2 | 4 | 0.015468 | 0.009067 | 1.705969 | 0.426492 |
| 128 * 128 | 2 | 16 | 0.017109 | 0.008509 | 2.010732 | 0.125671 |
| 256 * 256 | 4 | 64 | 0.162713 | 0.047969 | 3.392056 | 0.053001 |
|  |  |  |  |  |  |  |

**Graphs**:

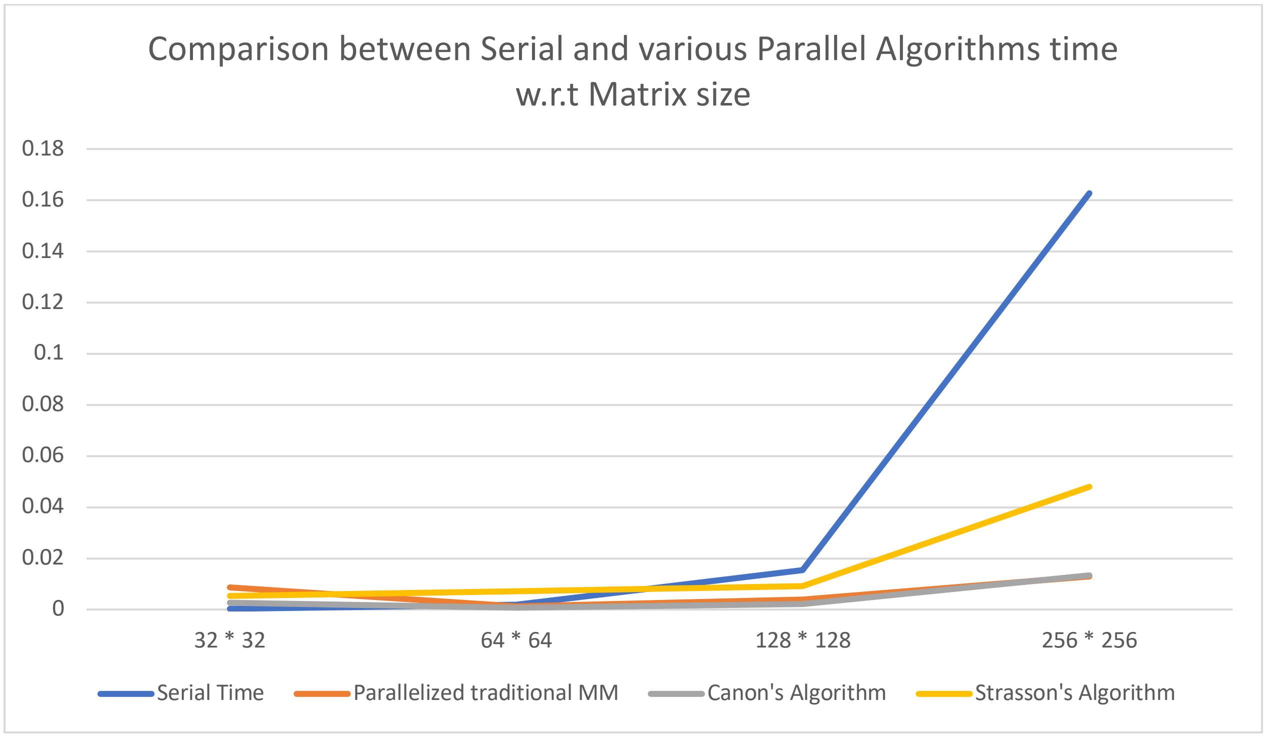


**FIGURE 1**

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

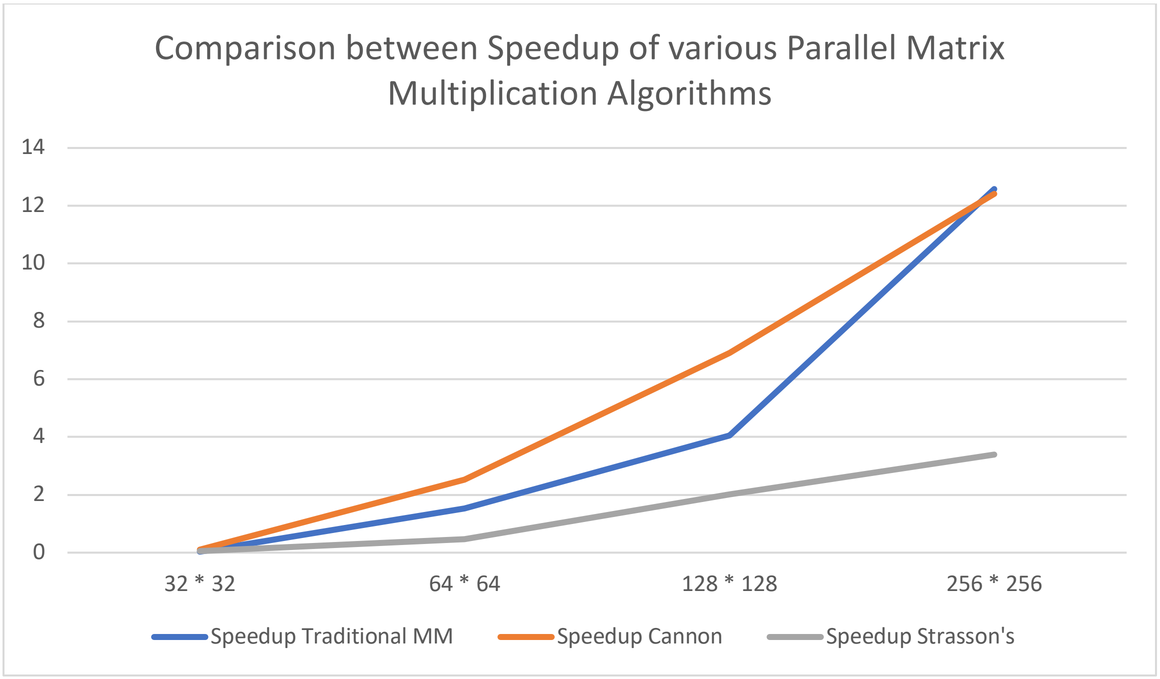


**FIGURE 2**

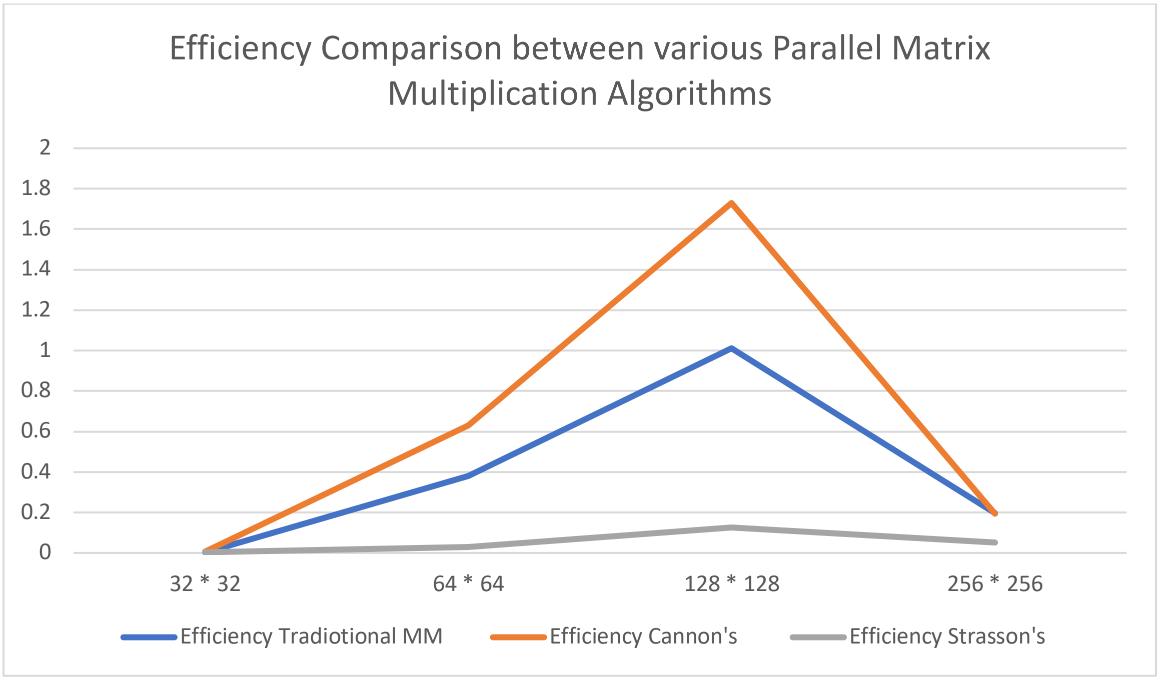


**FIGURE 3**

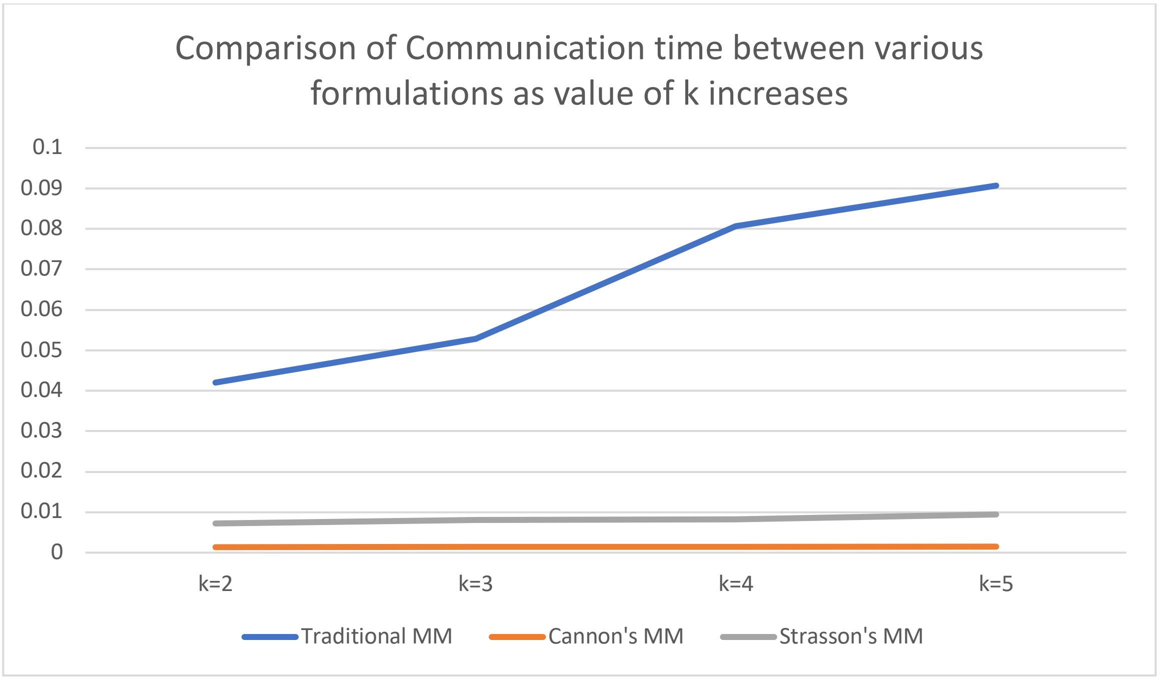


**FIGURE 4**

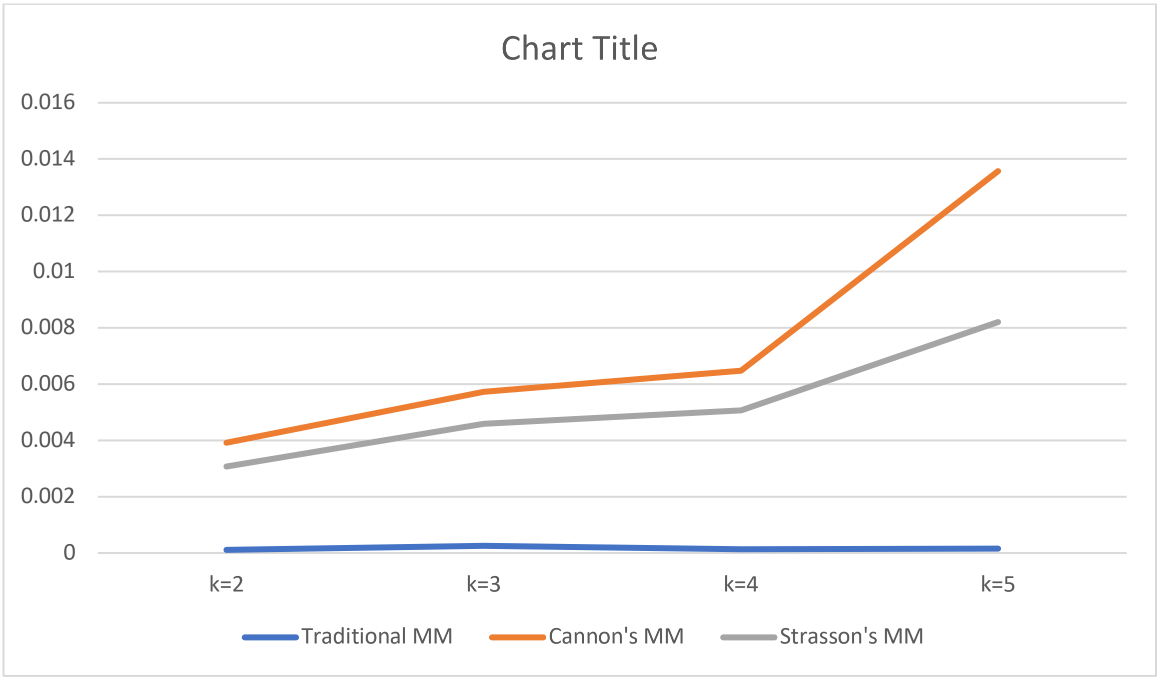


**FIGURE 5**

Aishwarya Ganesh: 665021069
Ankith C Kowshik: 678324283

**Analysis**:

1. The above graphs show the serial run time, parallel runtime, Speedup, and Efficiency of Parallelized traditional matrix multiplication, cannon's algorithm and strasson's algorithm.
2. From Figure 1, we can see the serial run time is almost similar (varies very little) compared to other parallel algorithms upto matrix size of 128*128 since the communication time between different processes in a parallel system nullify the advantage of parallel system. Thus the communication time is a overhead here which increases the overall time.
3. From Figure 1, we can observe that for a large matrix size of 256*256 the cannon's and strasson's algorithm peforms with lowest Tp time. Thus it is clear that the division of workload between processes in the parallel system saves time when they work in parallel.
4. From Figure 2 it is clear that Cannon's algorithm has the highest speedup.
5. Speedup increases with increase in matrix size since speedup is directly proportional to serial time which increases with increase in size of matrix.
6. Strasson's algorithm performance decreases with increase in processes due to implementation constraints.
7. Strasson's algorithm is memory intensive as well.
8. From Figure 3, we can see that efficiency drops as number of processes increase and matrix size increases. This is due to the overhead in parallel system.
9. Efficiency is directly proportional to speedup and inversely proportional to number of processes.
10. From Figure 4 it is evident that as k value increases, the workload increases hence the communicatin time also increases. The communication time is highest for the traditional parallelized Matrix Multiplication.
11. From Figure 5, we can see that the computational time also increases with k since there is increase in workload. It is highest for Cannon's implementation and then Strasson's.

**Lessons Learnt**:

1. In this lab assignment we learnt various formulations of matrix multiplication.
2. We understood the various factors that affects the time taken to compute matrix multiplication.
3. This assignment also helped understand how to decompose data in different formulations, and how to communicate between processes in these formulations.
4. In conclusion, parallelization decreases the time taken to calculate matrix multiplication however as matrix size increases, there is a overhead associated with communication time.
5. Time increases if we increase number of process nodes since communication time increases.
6. In this assignment we learnt how to calculate speedup and efficiency, the best parallel formulation that provides the best speedup and efficiency, and the factors that affects the two.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

//macros
const int N = 32; /* Size of matrix N * N */
int *coords;
//global variables
float *A;
/* Function to calulate the determinant of the matrix *
 * Input : matrix, size of matrix                     *
 * Output : determinant                              */
float find_det(float *local_matA, int N)
{
   int i, j, k;
   float det = 0;

   for (k=1; k<N; k++) {
      for (i=k+1; i<=N; i++) {
         for (j=k+1; j<=N; j++) {
            det = local_matA[i*N+j] * local_matA[k*N+k] - local_matA[i*N+k] * local_matA[k*N+j];
            if (k >= 2) {
               det = local_matA[i*N+j] / local_matA[k*N+k];
            }
         }
      }
   }
   return det;
}

float * Parallel_Multiply(float * A, float * B, float * C, int root, int my_rank, int no_of_processors, int rows, int columns) {
   float from, to;
   int i, j, k;
   float sum;
   if (rows % no_of_processors != 0) {
      if (my_rank == root) printf("Matrix size not divisible by number of processors\n");
      exit(-1);
   }
   int no_of_elements = rows * columns;
   float * result = (float * ) malloc(sizeof(float) * no_of_elements);
   from = my_rank * rows / no_of_processors;  // Calculating from
   to = (my_rank + 1) * rows / no_of_processors; // to

   MPI_Bcast(B, no_of_elements, MPI_DOUBLE, root, MPI_COMM_WORLD); // Broadcast entire matrix B to all processors
   if (my_rank == root)
 MPI_Scatter(A, no_of_elements / no_of_processors, MPI_DOUBLE, MPI_IN_PLACE, no_of_elements / no_of_processors,
MPI_DOUBLE, root, MPI_COMM_WORLD); // Scatter A (n/p) rows to all processors
   else
 MPI_Scatter(MPI_IN_PLACE, no_of_elements / no_of_processors, MPI_DOUBLE, A, no_of_elements / no_of_processors,
MPI_DOUBLE, root, MPI_COMM_WORLD); // Scatter A (n/p) rows to all processors

   int loop = (to - from);
   int ci = 0;
   for (i = from; i < to; i++) {
      for (j = 0; j < columns; j++) {
         sum = 0;
         for (k = 0; k < rows; k++) {
            sum += A[ci * rows + k] * B[k * rows + j]; // Calculate C = A*B
         }
         result[ci * rows + j] = sum;
      }
      ci++;
   }
   //Gather results to root.
```

```
    MPI_Gather(result, no_of_elements / no_of_processors, MPI_DOUBLE, C, no_of_elements / no_of_processors,
MPI_DOUBLE, root, MPI_COMM_WORLD);

    //free(result);
    return C;
}


/* Function that implements traditional matrix multiplication with time complexity O(n^3) *
 * Input : Matrixes to multiply, and size of matrix                                        *
 * Output : Result of Matrix Multiplication                                               */
float *trad_matrix_mul(float *local_matA, float *local_matB, int N)
{
    int i, j, k;
    float *res;
    res = (float*) malloc(sizeof(float) * (N*N));

    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            res[i*N+j] = 0;
            for (k=0; k<N; k++) {
                res[i*N+j] += local_matA[i*N+k] * local_matB[k*N+j];
            }
        }
    }
    return res;
}

float * Form_Matrix(float * C11, float * C12, float * C21, float * C22, float * C, int n) {

    int i, j, i2 = 0;
    int i3 = -n / 2;
    int i4 = i3;
    int j2, j3;

    for (i = 0; i < n; i++) {
        j2 = 0;
        j3 = 0;
        for (j = 0; j < n; j++) {

            if (j < (n / 2) && (i < n / 2)) // first Quadrant
                C[i * n + j] = C11[i * (n / 2) + j];

            if (j >= n / 2 && i < n / 2) {   //Second Quadrant
                C[i * n + j] = C12[i2 * (n / 2) + j2];
                j2++;
            }

            if (j < n / 2 && i >= n / 2) // Third quadrant.
                C[i * n + j] = C21[i3 * (n / 2) + j];

            if (i >= n / 2 && j >= n / 2) {
                C[i * n + j] = C22[i4 * (n / 2) + j3];
                j3++;
            }
        }

        i2++;
        i3++;
        i4++;
    }
    return C;
}

float *strasson_matrix_mul(float *A, MPI_Comm comm, int world_size, int rank)
{
```

```c
float *B, *C;
int rows, columns;
MPI_Datatype mysubarray, subarrtype;
MPI_Request req[8];
MPI_Status status;
int i,j;

int starts[2] = {
    0,
    0
};
//Upto to total length N
int bigns[2] = {
    N,
    N
};

int subns[2] = {
    N / 2, //rows
    N / 2 //columns
};

B = A;
C = (float*) malloc(sizeof(float) * (N*N));

MPI_Type_create_subarray(2, bigns, subns, starts, MPI_ORDER_C, MPI_FLOAT, & mysubarray);
MPI_Type_create_resized(mysubarray, 0, (N / 2) * (N / 2) * sizeof(float), & subarrtype);
MPI_Type_commit( & subarrtype);

rows = (N / 2);
columns = (N / 2);
int no_of_elements = rows * columns;
float * local_A = (float * ) malloc(no_of_elements * sizeof(float));
float * local_B = (float * ) malloc(no_of_elements * sizeof(float));
float * local_C = (float * ) malloc(no_of_elements * sizeof(float));

if (rank == 0) {

    MPI_Isend(A, 1, subarrtype, 0, 0, MPI_COMM_WORLD, & req[0]); /
    MPI_Irecv(local_A, no_of_elements, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, & req[0]);
    MPI_Wait( & req[0], & status);
    MPI_Isend(B, 1, subarrtype, 0, 0, MPI_COMM_WORLD, & req[0]);
    MPI_Irecv(local_B, no_of_elements, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, & req[0]);
    MPI_Wait( & req[0], & status);
    MPI_Isend(C, 1, subarrtype, 0, 0, MPI_COMM_WORLD, & req[0]);
    MPI_Irecv(local_C, no_of_elements, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, & req[0]);
    MPI_Wait( & req[0], & status);
    MPI_Isend(A + N / 2, 1, subarrtype, 1, 1, MPI_COMM_WORLD, & req[1]);
    MPI_Isend(B + N / 2, 1, subarrtype, 1, 1, MPI_COMM_WORLD, & req[1]);
    MPI_Isend(C + N / 2, 1, subarrtype, 1, 1, MPI_COMM_WORLD, & req[1]);

    MPI_Isend(A + ((N / 2) * (N)), 1, subarrtype, 2, 2, MPI_COMM_WORLD, & req[2]);
    MPI_Isend(B + ((N / 2) * (N)), 1, subarrtype, 2, 2, MPI_COMM_WORLD, & req[2]);
    MPI_Isend(C + ((N / 2) * (N)), 1, subarrtype, 2, 2, MPI_COMM_WORLD, & req[2]);

    MPI_Isend(A + ((N / 2) * N + (N - 1)), 1, subarrtype, 3, 3, MPI_COMM_WORLD, & req[3]);
    MPI_Isend(B + ((N / 2) * N + (N - 1)), 1, subarrtype, 3, 3, MPI_COMM_WORLD, & req[3]);
    MPI_Isend(C + ((N / 2) * N + (N - 1)), 1, subarrtype, 3, 3, MPI_COMM_WORLD, & req[3]);

}

if (rank == 1) {
    MPI_Irecv(local_A, no_of_elements, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, & req[1]);
    MPI_Irecv(local_B, no_of_elements, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, & req[1]);
    MPI_Irecv(local_C, no_of_elements, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, & req[1]);
    MPI_Wait( & req[1], & status);
}
```

```
    if (rank == 2) {
        MPI_Irecv(local_A, no_of_elements, MPI_FLOAT, 0, 2, MPI_COMM_WORLD, & req[2]);
        MPI_Irecv(local_B, no_of_elements, MPI_FLOAT, 0, 2, MPI_COMM_WORLD, & req[2]);
        MPI_Irecv(local_C, no_of_elements, MPI_FLOAT, 0, 2, MPI_COMM_WORLD, & req[2]);

        MPI_Wait( & req[2], & status);
    }
    if (rank == 3) {
        MPI_Irecv(local_A, no_of_elements, MPI_FLOAT, 0, 3, MPI_COMM_WORLD, & req[3]);
        MPI_Irecv(local_B, no_of_elements, MPI_FLOAT, 0, 3, MPI_COMM_WORLD, & req[3]);
        MPI_Irecv(local_C, no_of_elements, MPI_FLOAT, 0, 3, MPI_COMM_WORLD, & req[3]);
        MPI_Wait( & req[3], & status);
    }

    MPI_Type_free( & subarrtype);

    /* We completed sending the matrices to different processors */
    MPI_Barrier(MPI_COMM_WORLD);

    float * T1 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T2 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T3 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T4 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T5 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T6 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T7 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T8 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T9 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T10 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T11 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T12 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T13 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * T14 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * P1 = (float * ) malloc(sizeof(float) * (no_of_elements));
    float * P2 = (float * ) malloc(sizeof(float) * (no_of_elements));
    float * P3 = (float * ) malloc(sizeof(float) * (no_of_elements));
    float * P4 = (float * ) malloc(sizeof(float) * (no_of_elements));
    float * P5 = (float * ) malloc(sizeof(float) * (no_of_elements));
    float * P6 = (float * ) malloc(sizeof(float) * (no_of_elements));
    float * P7 = (float * ) malloc(sizeof(float) * (no_of_elements));

    if (T1 == NULL || T2 == NULL || T3 == NULL || T4 == NULL || T5 == NULL || T6 == NULL || T7 == NULL || T8 == NULL || T9
  == NULL || T10 == NULL || P1 == NULL || P2 == NULL || P3 == NULL || P4 == NULL)
        printf("Error allocating memory\n");

    if (rank == 0) {

        float * local_A22 = (float * ) malloc(sizeof(float) * no_of_elements);
        float * local_B22 = (float * ) malloc(sizeof(float) * no_of_elements);

        float * local_A12 = (float * ) malloc(sizeof(float) * no_of_elements);
        float * local_B21 = (float * ) malloc(sizeof(float) * no_of_elements);

        MPI_Isend(local_A, no_of_elements, MPI_FLOAT, 1, 100, MPI_COMM_WORLD, & req[0]);
        MPI_Isend(local_A, no_of_elements, MPI_FLOAT, 3, 500, MPI_COMM_WORLD, & req[4]);

        MPI_Isend(local_B, no_of_elements, MPI_FLOAT, 2, 400, MPI_COMM_WORLD, & req[3]);

        MPI_Isend(local_B, no_of_elements, MPI_FLOAT, 3, 600, MPI_COMM_WORLD, & req[4]);

        MPI_Irecv(local_A22, no_of_elements, MPI_FLOAT, 3, 100, MPI_COMM_WORLD, & req[0]);
        MPI_Irecv(local_B22, no_of_elements, MPI_FLOAT, 3, 200, MPI_COMM_WORLD, & req[0]);
        MPI_Wait( & req[0], & status);

        MPI_Irecv(local_A12, no_of_elements, MPI_FLOAT, 1, 100, MPI_COMM_WORLD, & req[1]);
    MPI_Wait( & req[1], & status);

    MPI_Irecv(local_B21, no_of_elements, MPI_FLOAT, 2, 200, MPI_COMM_WORLD, & req[2]);
    MPI_Wait( & req[0], & status);

    for (i = 0; i < no_of_elements; i++) {
        * (T1 + i) = * (local_A + i) + * (local_A22 + i);
        * (T2 + i) = * (local_B + i) + * (local_B22 + i);
        * (T3 + i) = * (local_A12 + i) - * (local_A22 + i);
        * (T4 + i) = * (local_B21 + i) + * (local_B22 + i);
    }

    local_A22 = NULL;
    local_B22 = NULL;
    local_A12 = NULL;
    local_B21 = NULL;

}

if (rank == 1) {
    float * local_A11 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * local_B22 = (float * ) malloc(sizeof(float) * no_of_elements);

    MPI_Isend(local_A, no_of_elements, MPI_FLOAT, 0, 100, MPI_COMM_WORLD, & req[1]);

    MPI_Isend(local_B, no_of_elements, MPI_FLOAT, 3, 500, MPI_COMM_WORLD, & req[1]);

    MPI_Irecv(local_A11, no_of_elements, MPI_FLOAT, 0, 100, MPI_COMM_WORLD, & req[0]);
    MPI_Irecv(local_B22, no_of_elements, MPI_FLOAT, 3, 200, MPI_COMM_WORLD, & req[0]);
    MPI_Wait( & req[0], & status);

    for (i = 0; i < no_of_elements; i++) {
        * (T5 + i) = * (local_B + i) - * (local_B22 + i);
        * (T6 + i) = * (local_A11 + i) + * (local_A + i);
        * (T7 + i) = * (local_B22 + i);
        * (T14 + i) = * (local_A11 + i);
    }


}

if (rank == 2) {

    float * local_A22 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * local_B11 = (float * ) malloc(sizeof(float) * no_of_elements);

    MPI_Isend(local_B, no_of_elements, MPI_FLOAT, 0, 200, MPI_COMM_WORLD, & req[2]);
    MPI_Isend(local_A, no_of_elements, MPI_FLOAT, 3, 500, MPI_COMM_WORLD, & req[2]);

    MPI_Irecv(local_A22, no_of_elements, MPI_FLOAT, 3, 100, MPI_COMM_WORLD, & req[3]);
    MPI_Wait( & req[3], & status);
    MPI_Irecv(local_B11, no_of_elements, MPI_FLOAT, 0, 400, MPI_COMM_WORLD, & req[3]);
    MPI_Wait( & req[3], & status);

    for (i = 0; i < no_of_elements; i++) {
        * (T8 + i) = * (local_A + i) + * (local_A22 + i);
        * (T9 + i) = * (local_B11 + i);
        * (T10 + i) = * (local_A22 + i);
        * (T11 + i) = * (local_B + i) - * (local_B11 + i);
    }

}

if (rank == 3) {

    float * local_A21 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * local_A11 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * local_B12 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * local_B11 = (float * ) malloc(sizeof(float) * no_of_elements);

    MPI_Isend(local_A, no_of_elements, MPI_FLOAT, 0, 100, MPI_COMM_WORLD, & req[0]);
    MPI_Isend(local_B, no_of_elements, MPI_FLOAT, 0, 200, MPI_COMM_WORLD, & req[0]);
    MPI_Isend(local_B, no_of_elements, MPI_FLOAT, 1, 200, MPI_COMM_WORLD, & req[0]);
    MPI_Isend(local_A, no_of_elements, MPI_FLOAT, 2, 100, MPI_COMM_WORLD, & req[3]);

    MPI_Irecv(local_A21, no_of_elements, MPI_FLOAT, 2, 500, MPI_COMM_WORLD, & req[4]);

    MPI_Irecv(local_B11, no_of_elements, MPI_FLOAT, 0, 600, MPI_COMM_WORLD, & req[4]);

    MPI_Irecv(local_B12, no_of_elements, MPI_FLOAT, 1, 500, MPI_COMM_WORLD, & req[4]);

    MPI_Irecv(local_A11, no_of_elements, MPI_FLOAT, 0, 500, MPI_COMM_WORLD, & req[5]);
    MPI_Wait( & req[4], & status);
    MPI_Wait( & req[5], & status);

    for (i = 0; i < no_of_elements; i++) {
      * (T12 + i) = * (local_A21 + i) - * (local_A11 + i);
      * (T13 + i) = * (local_B11 + i) + * (local_B12 + i);
    }

  }

  P1 = Parallel_Multiply(T1, T2, P1, 0, rank, world_size, rows, columns);
  P7 = Parallel_Multiply(T3, T4, P7, 0, rank, world_size, rows, columns);
  P3 = Parallel_Multiply(T14, T5, P3, 1, rank, world_size, rows, columns);
  P5 = Parallel_Multiply(T6, T7, P5, 1, rank, world_size, rows, columns);
  P2 = Parallel_Multiply(T8, T9, P2, 2, rank, world_size, rows, columns);
  P4 = Parallel_Multiply(T10, T11, P4, 2, rank, world_size, rows, columns);
  P6 = Parallel_Multiply(T12, T13, P6, 3, rank, world_size, rows, columns);

  /

  if (rank == 1) {
    MPI_Isend(P3, no_of_elements, MPI_FLOAT, 0, 100, MPI_COMM_WORLD, & req[3]);
    MPI_Isend(P5, no_of_elements, MPI_FLOAT, 0, 200, MPI_COMM_WORLD, & req[3]);

  }
  if (rank == 2) {
    MPI_Isend(P2, no_of_elements, MPI_FLOAT, 0, 300, MPI_COMM_WORLD, & req[4]);
    MPI_Isend(P4, no_of_elements, MPI_FLOAT, 0, 400, MPI_COMM_WORLD, & req[4]);

  }
  if (rank == 3) {
    MPI_Isend(P6, no_of_elements, MPI_FLOAT, 0, 500, MPI_COMM_WORLD, & req[4]);
  }
  MPI_Barrier(MPI_COMM_WORLD);



  if (rank == 0) {
    MPI_Irecv(P3, no_of_elements, MPI_FLOAT, 1, 100, MPI_COMM_WORLD, & req[3]);
    MPI_Irecv(P5, no_of_elements, MPI_FLOAT, 1, 200, MPI_COMM_WORLD, & req[3]);
    MPI_Irecv(P2, no_of_elements, MPI_FLOAT, 2, 300, MPI_COMM_WORLD, & req[4]);
    MPI_Irecv(P4, no_of_elements, MPI_FLOAT, 2, 400, MPI_COMM_WORLD, & req[4]);
    MPI_Irecv(P6, no_of_elements, MPI_FLOAT, 3, 500, MPI_COMM_WORLD, & req[5]);
    MPI_Wait( & req[3], & status);
    MPI_Wait( & req[4], & status);
    MPI_Wait( & req[5], & status);
    float * C11 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * C12 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * C21 = (float * ) malloc(sizeof(float) * no_of_elements);
    float * C22 = (float * ) malloc(sizeof(float) * no_of_elements);
```

```c
        for (i = 0; i < no_of_elements; i++) {
            * (C11 + i) = * (P1 + i) + * (P4 + i) - * (P5 + i) + * (P7 + i);
            * (C12 + i) = * (P3 + i) + * (P5 + i);
            * (C21 + i) = * (P2 + i) + * (P4 + i);
            * (C22 + i) = * (P1 + i) + * (P3 + i) - * (P2 + i) + * (P6 + i);
        }


        C = Form_Matrix(C11, C12, C21, C22, C, N);

    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    return C;
}

/* Function that implements matrix multiplication using Canon's Algorithm *
 * Input : Matrixes to multiply, and size of matrix                       *
 * Output : Result of Matrix Multiplication                              */
float *cannon_matrix_mul(float *local_matA, float *local_matB, int N, MPI_Comm comm)
{
    int i;
    int nlocal;
    int npes, dims[2], periods[2];
    int myrank, my2drank, mycoords[2], rightrank, leftrank, downrank, uprank;
    int shiftsource, shiftdest;
    float *res;
    res = (float*) malloc(sizeof(float) * (N*N));

    MPI_Status status;
    MPI_Comm comm_2d;

    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    dims[0] = dims[1] = sqrt(npes);
    periods[0] = periods[1] = 1; //for wraparound connenctions

    MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

    //Get the rank and coordinate with respect to new topology
    MPI_Comm_rank(comm_2d, &my2drank);
    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

    //Compute the rank of the up and left shifts
    MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
    MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

    nlocal = N/dims[0]; //determine the dimension of the local matrix block

    //Perform the initial matrix alignment first for A and then for B
    MPI_Cart_shift(comm_2d, 0, -1, &shiftsource, &shiftdest);
    MPI_Sendrecv_replace(local_matA, nlocal*nlocal, MPI_FLOAT, shiftdest, 1, shiftsource, 1, comm_2d, &status);

    MPI_Cart_shift(comm_2d, 1, -1, &shiftsource, &shiftdest);
    MPI_Sendrecv_replace(local_matB, nlocal*nlocal, MPI_FLOAT, shiftdest, 1, shiftsource, 1, comm_2d, &status);

    //Get into the main computaion loop
    for (i=0; i<dims[0]; i++) {
        res = trad_matrix_mul(local_matA, local_matB, nlocal);
        //shift matrix local_matA left by one
        MPI_Sendrecv_replace(local_matA, nlocal*nlocal, MPI_FLOAT, leftrank, 1, rightrank, 1, comm_2d, &status);
        //shift matrix local_matB up by one
        MPI_Sendrecv_replace(local_matB, nlocal*nlocal, MPI_FLOAT, uprank, 1, downrank, 1, comm_2d, &status);
    }

    //restore the original distribuation of local_matA and local_matB
    MPI_Cart_shift(comm_2d, 0, 1, &shiftsource, &shiftdest);
    MPI_Sendrecv_replace(local_matA, nlocal*nlocal, MPI_FLOAT, shiftdest, 1, shiftsource, 1, comm_2d, &status);
    MPI_Cart_shift(comm_2d, 1, 1, &shiftsource, &shiftdest);
    MPI_Sendrecv_replace(local_matB, nlocal*nlocal, MPI_FLOAT, shiftdest, 1, shiftsource, 1, comm_2d, &status);

    MPI_Comm_free(&comm_2d);
    return res;
}

/* Function to generate input for the matrix from the range of {-1, 0, 1} *
 * Input : NA                                                              *
 * Output : Matrix with entries in range {-1, 0, 1}                       */
float *gen_input()
{
    int i, j;
    float *matrix;
    int lo = -1, high = 1;

    matrix = (float*) malloc(sizeof(float) * (N*N));
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            matrix[i*N+j] = (int)(rand() % (high - lo + 1) + lo);
            //printf("%f\n",matrix[i*N+j]);
        }
    }
    return matrix;
}

/* Function to send matrix over mesh topology        *
 * Input : comm, rank, world_size, local size of n   *
 * Output : local matrix A                           */
float *send_mat_mesh(MPI_Comm *comm, int rank, int world_size, int *nlocal)
{
    float *local_matA;
    float *buffer;
    int i, j, k, l;
    int index, row, col;
    int p = (int) sqrt(world_size);
    int n_proc = N / p;
    MPI_Status status;

    local_matA = (float*) malloc(sizeof(float) * n_proc * n_proc);
    if (rank == 0)
    {
        A = gen_input(world_size, rank);

        for (i=0; i<p; i++) {
            for (j=0; j<p; j++) {
                if (i==0 && j==0) {
                    index = 0;
                    for (k=0; k<n_proc; k++) {
                        for (l=0; l<n_proc; l++) {
                            local_matA[index++] = A[k*N+1];
                        }
                    }
                }
                else {
                    buffer = (float*) malloc(sizeof(float) * (n_proc * n_proc));
                    row = i * n_proc;
                    col = j * n_proc;
                    index = 0;

                    /* Filling the buffer to send to process */
                    for (k=row; k<row+n_proc; k++) {
                        for (l=col; l<col+n_proc; l++) {
                            buffer[index++] = A[k*N+1];
                        }
                    }
                    MPI_Send(buffer, n_proc*n_proc, MPI_FLOAT, j*p+i,0, *comm);
                    free(buffer);
                }
            }
        }
    }
    else {
        MPI_Recv(local_matA, n_proc*n_proc, MPI_FLOAT,0,0, *comm, &status);
        //printf("MPI Status %d %d %d\n", status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR);
    }
    return local_matA;
}

void create_2dmesh_topo(MPI_Comm *comm, int *rank, int *world_size)
{
    MPI_Comm_size(MPI_COMM_WORLD, world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, rank);
    int dims[2], periods[2];
    coords = (int*) malloc(sizeof(int) * 2);
    dims[0] = dims[1] = (int) sqrt((float) *world_size);
    periods[0] = periods[1] = 1;

    int rv = MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, comm);
    //printf("rv is %d\n",rv);
    //MPI_Comm_size(*comm, world_size);
    MPI_Cart_coords(*comm, *rank, 2, coords);
    //printf("world size %d rank %d\n", *world_size, *rank);
}


int main(int argc, char** argv)
{
    int world_size, rank, i, k = 4, nlocal;
    float *local_matA, *local_matB, *res, *s;
    float det[1], result[1];

    MPI_Status status;
    MPI_Comm comm;
    //Initialize the MPI environment
    MPI_Init(&argc, &argv);
    double start , end, Ts, mesh_traditional_time, mesh_cannon_time, mesh_strasson_time;
  double Speedup, Eff;
    //get the total number of processors
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0) {
 start = MPI_Wtime();
        printf("The total number of processors: %d\n", world_size);
        printf("The size of the matrix - %d x %d\n", N, N);
        printf("The value of k - %d\n", k);
 A = gen_input();
 trad_matrix_mul(A, A, N);
        det[0] = find_det(A,N);
 end = MPI_Wtime();
 Ts = end - start;
 printf("Serial Time is %lf\n",Ts);
    }

    start = MPI_Wtime();
    create_2dmesh_topo(&comm, &rank, &world_size);
    nlocal = N / sqrt(world_size);

      local_matA = send_mat_mesh(&comm, rank, world_size, &nlocal);

    /* Set 1 : traditional O(n^3) MM Algorithm */
    if (k == 1) { //if k=1, find the determinant directly, no matrix multiplication
        det[0] = find_det(local_matA, nlocal);
        MPI_Reduce(&det, &result, 1, MPI_FLOAT, MPI_PROD, 0, comm);
        if (rank == 0) {
            printf("Rank %d The determinant of the matrix using 2D mesh mapping and O(n^3) MM Algo is: %f\n", rank, result[0]);
        }
    } else { //if k>1, do the matrix multiplication then find the determinant
        local_matB = local_matA;
        for (i=0; i<k; i++) {
            res = trad_matrix_mul(local_matA, local_matB, nlocal);
        }

        det [0] = find_det(res, nlocal);
        MPI_Reduce(&det, &result, 1, MPI_FLOAT, MPI_PROD, 0, comm);
        if (rank == 0) {
            printf("The determinant of the matrix using 2D mesh mapping and O(n^3) MM Algo is: %f\n", result[0]);
        }
    }
    end = MPI_Wtime();
    mesh_traditional_time = end - start; //calculate the total execuation time for the mesh mapping

      if (rank == 0) {
    printf("\nThe total time is needed to find the determinant using 2D mesh mapping and n^3 algorithm: %lf\n",
mesh_traditional_time);
 Speedup = Ts/mesh_traditional_time;
 Eff = Speedup/world_size;
 printf("S %lf E %lf\n",Speedup,Eff);
 }
/* Set 2: Cannon's method */
      start = MPI_Wtime();
      if (k>1) {
      local_matA = send_mat_mesh(&comm, rank, world_size, &nlocal);
      local_matB = local_matA;
      for (i=0; i<k; i++) {
      res = cannon_matrix_mul(local_matA, local_matB, nlocal, comm);
      }
      det[0] = find_det(res, nlocal);
      MPI_Reduce(&det, &result, 1, MPI_FLOAT, MPI_PROD, 0, comm);
      if (rank == 0) {
      printf("\nThe determinant of the matrix using 2D mesh mapping and cannon's method is: %f\n", result[0]);
      }
      }
      end = MPI_Wtime();
      mesh_cannon_time = end - start; //calculate the total execuation time for the cannon's method
      if (rank == 0) {
       printf("\nThe total time is needed to find the determinant using 2D mesh mapping and cannon's method: %lf\n",
mesh_cannon_time);
 Speedup = Ts/mesh_cannon_time;
 Eff = Speedup/world_size;
 printf("S %lf E %lf\n",Speedup,Eff);
 }

    /* Set 3: Strasson's method */
      start = MPI_Wtime();
      if (k>1) {
      for (i=0; i<k; i++) {
      res = strasson_matrix_mul(A, comm, world_size, rank);
      }
      det[0] = find_det(res, nlocal);
      MPI_Reduce(&det, &result, 1, MPI_FLOAT, MPI_PROD, 0, comm);
      if (rank == 0) {
        printf("\nThe determinant of the matrix using 2D mesh mapping and cannon's method is: %lf\n", result[0]);
        }
        }
        end = MPI_Wtime();
        mesh_strasson_time = end - start; //calculate the total execuation time for the cannon's method
        if (rank == 0) {
         printf("\nThe total time is needed to find the determinant using 2D mesh mapping and strasson's method: %f\n",
mesh_strasson_time);
 Speedup = Ts/mesh_strasson_time;
 Eff = Speedup/world_size;
 printf("S %lf E %lf\n",Speedup,Eff);
 }
    //free(local_matA);
    //free(A);
    MPI_Comm_free(&comm);
    MPI_Finalize();

    return 0;
}
```