ASSIGNMENT 1

Name – Sarvesh Sanjay Shingare
Roll No – 144
PRN – 202101050031
Batch – B4

✷ Problem Statement :- Implement Cannon's Matrix-Matrix Multiplication Algorithm using MPI making use of multiple cores.

Theory :-

Cannon's algorithm is a well-known technique for parallel matrix multiplication, specifically designed to run efficiently on distributed systems where multiple processors are available. The primary aim of this algorithm is to break down the large matrix multiplication task into smaller, manageable sub-blocks that can be processed simultaneously across multiple cores, leading to faster computation.

Key Concepts:

1. Process Grid and Distribution: In Cannon's algorithm, the processors are arranged in a 2D grid (like a chessboard), where each processor handles a sub-block of the matrices involved. The grid is typically square (e.g., 2x2 for 4 processors). The original large matrices are split into smaller blocks, and these blocks are distributed across the processors.
2. Initial Alignment: Before starting the multiplication, each block of matrix A is shifted left by its row index, and each block of matrix B is shifted up by its column index. This alignment ensures that each processor starts with the correct blocks for multiplication.
3. Iterative Multiplication and Shifting: The algorithm operates in multiple steps (iterations), where:
   o Each processor performs a multiplication of its current sub-blocks of matrices A and B.
   o After each multiplication, the blocks of matrix A are cyclically shifted left, and the blocks of matrix B are shifted up. This ensures that every processor eventually works with every required sub-block to compute its part of the final result.
4. Final Aggregation: After completing all iterations, each processor holds a portion of the final product matrix. The results are then gathered and combined to produce the complete matrix multiplication output.
   Applying the Concept with MPI:
   MPI (Message Passing Interface) is the perfect tool to implement Cannon's algorithm because it provides the necessary communication between processors. The program uses functions like MPI_Cart_create to set up the grid, MPI_Cart_shift for shifting the blocks, and MPI_Gather to collect the results. The code demonstrates efficient use of parallelism, proper block alignment, and the use of MPI for communication, which are critical for large-scale matrix operations in a distributed environment.

Code :-

```
// Sarvesh Shingare
// 202101050031

#include <mpi.h>
#include <iostream>
#include <vector>
#include <cmath>

#define N 4                 // Matrix size (NxN)
#define BLOCK_SIZE (N / 2) // Assuming a 2x2 process grid

void multiplyMatrices(const std::vector<int> &A, const std::vector<int> &B, std::vector<int> &C
{
    for (int i = 0; i < BLOCK_SIZE; ++i)
    {
        for (int j = 0; j < BLOCK_SIZE; ++j)
        {

            for (int k = 0; k < BLOCK_SIZE; ++k)
            {
                C[i * BLOCK_SIZE + j] += A[i * BLOCK_SIZE + k] * B[k * BLOCK_SIZE + j];
            }
        }
    }
}
int main(int argc, char \*_argv)
{
    int rank, size, sqrt_p;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    sqrt_p = std::sqrt(size);
    if (sqrt_p _ sqrt * p != size || N % sqrt_p != 0)
    {
        if (rank == 0)
        {
            std::cerr << "Error: The number of processes must be a perfect square and divisible
        }
        MPI_Finalize();
        return -1;
    }
```

```
int subMatSize = N / sqrt_p;
std::vector<int> A(subMatSize * subMatSize, 0);
std::vector<int> B(subMatSize _ subMatSize, 0);
std::vector<int> C(subMatSize _ subMatSize, 0); // Result matrix
                                                // Initialize matrices A and B
for (int i = 0; i < subMatSize \_ subMatSize; ++i)
{
    A[i] = 1; // Example initialization
    B[i] = 1; // Example initialization
}

MPI_Comm gridComm;
int dims[2] = {sqrt_p, sqrt_p};
int periods[2] = {1, 1}; // Enable wrap-around (cyclic shifts)
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &gridComm);

int coords[2];
MPI_Cart_coords(gridComm, rank, 2, coords);

int left, right, up, down;
MPI_Cart_shift(gridComm, 1, -1, &right, &left);
MPI_Cart_shift(gridComm, 0, -1, &down, &up);

// Initial alignment for A and B
MPI *Sendrecv_replace(&A[0], subMatSize * subMatSize, MPI * INT, left, 0, right, 0, gridComm
MPI_Sendrecv_replace(&B[0], subMatSize * subMatSize, MPI_INT, up, 0, down,
                     0, gridComm, MPI_STATUS_IGNORE);

// Perform Cannon's algorithm iterations for (int i = 0; i < sqrt_p; ++i)
{
    multiplyMatrices(A, B, C);

    // Shift left A
    MPI_Sendrecv_replace(&A[0], subMatSize * subMatSize, MPI_INT, left, 0, right, 0, gridCom
    // Shift up B
    MPI_Sendrecv_replace(&B[0], subMatSize * subMatSize, MPI_INT, up, 0, down,

                         0, gridComm, MPI_STATUS_IGNORE);
}

// Gather results at rank 0 std::vector<int> finalResult; if (rank == 0)
{
    finalResult.resize(N _ N);
```

```
    }
    MPI_Gather(&C[0], subMatSize * subMatSize, MPI_INT, &finalResult[0], subMatSize \* subMatSi

    if (rank == 0)
    {
        std::cout << "Result matrix:" << std::endl;
        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                std::cout << finalResult[i * N + j] << " ";
            }
            std::cout << std::endl;
        }
    }

    MPI_Finalize();
    return 0;
}
```

Output :-