THỰC HÀNH HPC BUỔI 4

Nội dung: HPC với MPI

Yêu cầu:

- Sinh viên thực hiện nghiêm túc các yêu cầu của từng bài
- Báo cáo nộp yêu cầu ngắn gọn: Trả lời chính xác các yêu cầu của mỗi bài

Bài 1: Nhân ma trận với MPI

a) Dùng hàm MPI_send/MPI_recv

- Xem lại hàm MPI_send()/MPI_recv(), các tham số của hàm
- Cơ chế truyền thông point- to- point, Blocking
- Nhập chương trình sau và chạy
- Phân tích cấu trúc chương trình và xây dựng lại thuật toán nhân ma trận từ chương trình.
- So sánh thời gian chạy chương trình với trường hợp chạy tuần tự và lập bảng với kích cỡ ma trận khác nhau.

//code

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define SIZE 4
#define FROM_MASTER 1
#define FROM_WORKER 2
#define DEBUG 1

MPI_Status status;

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void init_matrix(void)
{
    int i, j;
    for (i = 0; i < SIZE; i++)
    {
        for (j = 0; j < SIZE; j++) {
            a[i][j] = 1;
            b[i][j] = 1;

        } //end for i
    }  //end for j
} //end init_matrix()
```

```
static void print_matrix(void)
{
    int i, j;
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf("%7.2f", c[i][j]);
        } //end for i
        printf("\n");
    }    //end for j
}        //end print_matrix

int main(int argc, char** argv)
{
    int myrank, nproc;
    int rows;
    int mtype;
    int dest, src, offseta, offsetb;
    double start_time, end_time;
    int i, j, k, l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    rows = SIZE / nproc;  //compute the block size
    mtype = FROM_MASTER; //  =1

    if (myrank == 0) {
        /*Initialization*/
        printf("SIZE = %d, number of nodes = %d\n", SIZE, nproc);
        init_matrix();

        start_time = MPI_Wtime();

        if (nproc == 1) {
            for (i = 0; i < SIZE; i++) {
                for (j = 0; j < SIZE; j++) {
                    for (k = 0; k < SIZE; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[j][k];
                } //end for i
            }  //end for j
            end_time = MPI_Wtime();
            print_matrix();//--------------------------------
            printf("Execution time on %2d nodes: %f\n", nproc, end_time -
                start_time);
        } // end  if(nproc == 1)

        else {

            for (l = 0; l < nproc; l++) {
                offsetb = rows * l;  //start from (block size * processor id)
                offseta = rows;
                mtype = FROM_MASTER; // tag =1
```

```
                for (dest = 1; dest < nproc; dest++) {
                    MPI_Send(&offseta, 1, MPI_INT, dest, mtype,
                        MPI_COMM_WORLD);
                    MPI_Send(&offsetb, 1, MPI_INT, dest, mtype,
                        MPI_COMM_WORLD);
                    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                    MPI_Send(&a[offseta][0], rows * SIZE, MPI_DOUBLE, dest,
                        mtype, MPI_COMM_WORLD);
                    MPI_Send(&b[0][offsetb], rows * SIZE, MPI_DOUBLE, dest,
                        mtype, MPI_COMM_WORLD);

                    offseta += rows;
                    offsetb = (offsetb + rows) % SIZE;

                } // end for dest

                offseta = rows;
                offsetb = rows * l;

                //--mult the final local and print final global mult
                for (i = 0; i < offseta; i++) {
                    for (j = offsetb; j < offsetb + rows; j++) {
                        for (k = 0; k < SIZE; k++) {
                            c[i][j] = c[i][j] + a[i][k] * b[k][j];
                        }//end for k
                    } //end for j
                }// end for i
                   /*- wait for results from all worker tasks */
                mtype = FROM_WORKER;
                for (src = 1; src < nproc; src++) {
                    MPI_Recv(&offseta, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                        &status);
                    MPI_Recv(&offsetb, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                        &status);
                    MPI_Recv(&rows, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                        &status);
                    for (i = 0; i < rows; i++) {
                        MPI_Recv(&c[offseta + i][offsetb], offseta,
MPI_DOUBLE,src, mtype, MPI_COMM_WORLD, &status);
                    } //end for scr
                }//end for i
            } //end for l
            end_time = MPI_Wtime();
            print_matrix();
            printf("Execution time on %2d nodes: %f\n", nproc, end_time -
                start_time);
        }//end else
    } //end if (myrank == 0)

    else {
        /*---------------------------- worker----------------------*/
        if (nproc > 1) {
            for (l = 0; l < nproc; l++) {
                mtype = FROM_MASTER;
                MPI_Recv(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
```

```
                    &status);
                MPI_Recv(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                    &status);
                MPI_Recv(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                    &status);

                MPI_Recv(&a[offseta][0], rows * SIZE, MPI_DOUBLE, 0, mtype,
                    MPI_COMM_WORLD, &status);
                MPI_Recv(&b[0][offsetb], rows * SIZE, MPI_DOUBLE, 0, mtype,
                    MPI_COMM_WORLD, &status);

                for (i = offseta; i < offseta + rows; i++) {
                    for (j = offsetb; j < offsetb + rows; j++) {
                        for (k = 0; k < SIZE; k++) {
                            c[i][j] = c[i][j] + a[i][k] * b[k][j];
                        } //end for j
                    } //end for i

                } //end for l

                mtype = FROM_WORKER;
                MPI_Send(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
                MPI_Send(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
                MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
                for (i = 0; i < rows; i++) {
                    MPI_Send(&c[offseta + i][offsetb], offseta, MPI_DOUBLE, 0,
                        mtype, MPI_COMM_WORLD);

                } //end for i
            }//end for l

        } //end if (nproc > 1)
    } // end else

    MPI_Finalize();
    return 0;
} //end main(
```

b) Chương trình tham khảo

//code

```
#include<stdio.h>
#include<mpi.h>
#define NUM_ROWS_A 12 //rows of input [A]
#define NUM_COLUMNS_A 12 //columns of input [A]
#define NUM_ROWS_B 12 //rows of input [B]
#define NUM_COLUMNS_B 12 //columns of input [B]
#define MASTER_TO_SLAVE_TAG 1 //tag for messages sent from master to slaves
#define SLAVE_TO_MASTER_TAG 4 //tag for messages sent from slaves to master
void makeAB(); //makes the [A] and [B] matrixes
void printArray(); //print the content of output matrix [C];

int rank; //process rank
int size; //number of processes
int i, j, k; //helper variables
double mat_a[NUM_ROWS_A][NUM_COLUMNS_A]; //declare input [A]
double mat_b[NUM_ROWS_B][NUM_COLUMNS_B]; //declare input [B]
```

```
double mat_result[NUM_ROWS_A][NUM_COLUMNS_B]; //declare output [C]
double start_time; //hold start time
double end_time; // hold end time
int low_bound; //low bound of the number of rows of [A] allocated to a slave
int upper_bound; //upper bound of the number of rows of [A] allocated to a slave
int portion; //portion of the number of rows of [A] allocated to a slave
MPI_Status status; // store status of a MPI_Recv
MPI_Request request; //capture request of a MPI_Isend

int main(int argc, char* argv[])
{

    MPI_Init(&argc, &argv); //initialize MPI operations
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get the rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes

    /* master initializes work*/
    if (rank == 0) {
        makeAB();
        start_time = MPI_Wtime();
        for (i = 1; i < size; i++) {//for each slave other than the master
            portion = (NUM_ROWS_A / (size - 1)); // calculate portion without master
            low_bound = (i - 1) * portion;
            if (((i + 1) == size) && ((NUM_ROWS_A % (size - 1)) != 0)) {//if rows of
[A] cannot be equally divided among slaves
                upper_bound = NUM_ROWS_A; //last slave gets all the remaining rows
            }
            else {
                upper_bound = low_bound + portion; //rows of [A] are equally
divisable among slaves
            }
            //send the low bound first without blocking, to the intended slave
            MPI_Isend(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG,
MPI_COMM_WORLD, &request);
            //next send the upper bound without blocking, to the intended slave
            MPI_Isend(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &request);
            //finally send the allocated row portion of [A] without blocking, to the
intended slave
            MPI_Isend(&mat_a[low_bound][0], (upper_bound - low_bound) *
NUM_COLUMNS_A, MPI_DOUBLE, i, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &request);
        }
    }
    //broadcast [B] to all the slaves
    MPI_Bcast(&mat_b, NUM_ROWS_B * NUM_COLUMNS_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* work done by slaves*/
    if (rank > 0) {
        //receive low bound from the master
        MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&status);
        //next receive upper bound from the master
        MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &status);
        //finally receive row portion of [A] to be processed from the master
        MPI_Recv(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A,
MPI_DOUBLE, 0, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);
```

```
        for (i = low_bound; i < upper_bound; i++) {//iterate through a given set of
rows of [A]
            for (j = 0; j < NUM_COLUMNS_B; j++) {//iterate through columns of [B]
                for (k = 0; k < NUM_ROWS_B; k++) {//iterate through rows of [B]
                    mat_result[i][j] += (mat_a[i][k] * mat_b[k][j]);
                }
            }
        }
        //send back the low bound first without blocking, to the master
        MPI_Isend(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&request);
        //send the upper bound next without blocking, to the master
        MPI_Isend(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD, &request);
        //finally send the processed portion of data without blocking, to the master
        MPI_Isend(&mat_result[low_bound][0], (upper_bound - low_bound) *
NUM_COLUMNS_B, MPI_DOUBLE, 0, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &request);
    }

    /* master gathers processed work*/
    if (rank == 0) {
        for (i = 1; i < size; i++) {// untill all slaves have handed back the
processed data
            //receive low bound from a slave
            MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&status);
            //receive upper bound from a slave
            MPI_Recv(&upper_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD, &status);
            //receive processed data from a slave
            MPI_Recv(&mat_result[low_bound][0], (upper_bound - low_bound) *
NUM_COLUMNS_B, MPI_DOUBLE, i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
        }
        end_time = MPI_Wtime();
        printf("\nRunning Time = %f\n\n", end_time - start_time);
        printArray();
    }
    MPI_Finalize(); //finalize MPI operations
    return 0;
}

void makeAB()
{
    for (i = 0; i < NUM_ROWS_A; i++) {
        for (j = 0; j < NUM_COLUMNS_A; j++) {
            mat_a[i][j] = i + j;
        }
    }
    for (i = 0; i < NUM_ROWS_B; i++) {
        for (j = 0; j < NUM_COLUMNS_B; j++) {
            mat_b[i][j] = i * j;
        }
    }
}

void printArray()
{
    for (i = 0; i < NUM_ROWS_A; i++) {
```

```
            printf("\n");
            for (j = 0; j < NUM_COLUMNS_A; j++)
                printf("%8.2f  ", mat_a[i][j]);
        }
        printf("\n\n\n");
        for (i = 0; i < NUM_ROWS_B; i++) {
            printf("\n");
            for (j = 0; j < NUM_COLUMNS_B; j++)
                printf("%8.2f  ", mat_b[i][j]);
        }
        printf("\n\n\n");
        for (i = 0; i < NUM_ROWS_A; i++) {
            printf("\n");
            for (j = 0; j < NUM_COLUMNS_B; j++)
                printf("%8.2f  ", mat_result[i][j]);
        }
        printf("\n\n");
}
```

c) Nhân ma trận với hàm Scatterv/ Gatherv with "large" 2D matrices

- Xem lại hàm MPI_Scattery/MPI_Gathery, các tham số của hàm
- Cơ chế truyền thông của MPI_Scattery/MPI_Gathery
- Nhập chương trình sau và chạy
- Phân tích cấu trúc chương trình và xây dựng lại thuật toán nhân ma trận từ chương trình, vẽ biểu đồ thể hiện hoạt động của các hàm Scattery/MPI_Gathery trong chương trình.
- So sánh thời gian chạy chương trình với trường hợp chạy tuần tự và lập bảng với kích cỡ ma trận khác nhau.

//code

```
#include "mpi.h"
#include <stdio.h>
#define N 184 // grid size
#define procN 2  // size of process grid

int main(int argc, char** argv) {
    double* gA = nullptr; // pointer to array
    int rank, size;       // rank of current process and no. of processes

    // mpi initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // force to use correct number of processes
    if (size != procN * procN) {
        if (rank == 0) fprintf(stderr, "%s: Only works with np = %d.\n", argv[0],
procN * procN);
        MPI_Abort(MPI_COMM_WORLD, 1);
```

```
    }

    // allocate and print global A at master process
    if (rank == 0) {
        gA = new double[N * N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                gA[j * N + i] = j * N + i;
            }
        }

        printf("A is:\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%f ", gA[j * N + i]);
            }
            printf("\n");
        }
    }

    // create local A on every process which we'll process
    double* lA = new double[N / procN * N / procN];

    // create a datatype to describe the subarrays of the gA array
    int sizes[2] = { N, N }; // gA size
    int subsizes[2] = { N / procN, N / procN }; // lA size
    int starts[2] = { 0,0 }; // where this one starts
    MPI_Datatype type, subarrtype;
    MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE,
&type);
    MPI_Type_create_resized(type, 0, N / procN * sizeof(double), &subarrtype);
    MPI_Type_commit(&subarrtype);

    // compute number of send blocks
    // compute distance between the send blocks
    int sendcounts[procN * procN];
    int displs[procN * procN];

    if (rank == 0) {
        for (int i = 0; i < procN * procN; i++) {
            sendcounts[i] = 1;
        }
        int disp = 0;
        for (int i = 0; i < procN; i++) {
            for (int j = 0; j < procN; j++) {
                displs[i * procN + j] = disp;
                disp += 1;
            }
            disp += ((N / procN) - 1) * procN;
        }
    }

    // scatter global A to all processes
    MPI_Scatterv(gA, sendcounts, displs, subarrtype, lA,
        N * N / (procN * procN), MPI_DOUBLE,
        0, MPI_COMM_WORLD);

    // print local A's on every process
```

```
for (int p = 0; p < size; p++) {
    if (rank == p) {
        printf("la on rank %d:\n", rank);
        for (int i = 0; i < N / procN; i++) {
            for (int j = 0; j < N / procN; j++) {
                printf("%f ", lA[j * N / procN + i]);
            }
            printf("\n");
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);

// write new values in local A's
for (int i = 0; i < N / procN; i++) {
    for (int j = 0; j < N / procN; j++) {
        lA[j * N / procN + i] = rank;
    }
}

// gather all back to master process
MPI_Gatherv(lA, N * N / (procN * procN), MPI_DOUBLE,
    gA, sendcounts, displs, subarrtype,
    0, MPI_COMM_WORLD);

// print processed global A of process 0
if (rank == 0) {
    printf("Processed gA is:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", gA[j * N + i]);
        }
        printf("\n");
    }
}

MPI_Type_free(&subarrtype);

if (rank == 0) {
    delete gA;
}

delete lA;

MPI_Finalize();

return 0;
```

d) Nhân ma trận với MPI_Bcast, MPI_Scatter, and MPI_Gather.

- Xem lại cấu trúc hàm `MPI_Bcast, MPI_Scatter, and MPI_Gather` các tham số của hàm

- Nhập chương trình sau và chạy

- Phân tích cấu trúc chương trình và xây dựng lại thuật toán nhân ma trận từ chương trình.

- So sánh thời gian chạy chương trình với trường hợp chạy tuần tự và lập bảng với kích cỡ ma trận khác nhau.

//code

```
#include "mpi.h"
#include <stdio.h>

#define WIDTH 10        /* the size of the vector */

int main(int argc, char* argv[]) {

    int nprocs,         /* number of processes */
        rank,           /* rank of each process */
        chunk_size,     /* number of rows to be sent to a process */
        i, j;

    int matrix[WIDTH][WIDTH];           /* matrix for multiplication */
    int vector[WIDTH];                  /* vector for multiplication */
    int local_matrix[WIDTH][WIDTH];     /* for storing piece of matrix in each
process */
    int result[WIDTH];                  /* for storing result in each process */
    int global_result[WIDTH];           /* vector result after all calculations */

    /* Initialize MPI execution environment */
    // TO DO:
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // end TO DO

    printf("Hello from process %d of %d \n", rank, nprocs);

    chunk_size = WIDTH / nprocs;          /* number of rows to be sent to each
process */

    /* master doing part of the work here */
    if (rank == 0) {

        /* Initialize Matrix and Vector */
        for (i = 0; i < WIDTH; i++) {
            // Change here if you want to use random integer
            vector[i] = 1;
            for (j = 0; j < WIDTH; j++) {
                // Change here if you want to use random integer
                matrix[i][j] = 1;
            }
        }
    }

    /* Distribute Vector */
    /* All processes need the input vector for multiplication, so we can broadcast
it to all processes */
```

```
    // TO DO:
    MPI_Bcast(vector, WIDTH, MPI_INT, 0, MPI_COMM_WORLD);

    /* Distribute Matrix */
    /* We can broadcast entire matrix to all processes, but matrix might be too big,
     * and it is not an efficient method. Instead we will split matrix by row.
     * Scatter will do that for us. It will take the matrix, and send WIDTH *
chunk_size
     * to each process, and each piece get stored in local_matrix.
     */
     // TO DO:
    MPI_Scatter(matrix, WIDTH * chunk_size, MPI_INT, local_matrix, WIDTH *
chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

    /* Each processor has some rows of matrix, and the vector. Each process works on
their multiplication
     * and storing the result in result vector.
     */
    for (i = 0; i < chunk_size; i++) {
        result[i] = 0;
        for (j = 0; j < WIDTH; j++) {
            result[i] += local_matrix[i][j] * vector[j];
        }
    }

    /* Each process sends result back to master, and get stored in global_result */
    // TO DO:
    MPI_Gather(result, chunk_size, MPI_INT, global_result, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    /* master prints elements of the result */
    if (rank == 0) {
        for (i = 0; i < WIDTH; i++) {
            printf("%d\n", global_result[i]);
        }
    }
    /* Terminate MPI execution environment */
    MPI_Finalize();
    return 0;
}
```

Bài 2: Nhân ma trận với thuật toán Cannon với MPI

- Nghiên cứu thuật toán nhân ma trận Cannon
- Lập trình nhân ma trận với thuật toán Cannon với MPI
- So sánh thời gian chạy song song với trường hợp chạy tuần tự, tính hệ số speedup, hệ số hiệu quả và só sánh với kết quả tính theo lý thuyết, giải thích.
- Đối sánh lại mã chương trình với thuật toán nhân ma trận Cannon.

//Code of Cannon' Algorithm

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
```

```
#include <iostream>
using namespace std;
int main(int argc, char** argv)
{
    int x = 0;
    double kk;
    int proces;
    int numprocs;
    int right_neigh, left_neigh, up_neigh, down_neigh;
    int tag = 99;

    static const int n = 6; //size of matrices

    int psa[n][n]; //nxn
    int psb[n][n];
    int pra[n][n];
    int prb[n][n];
    int c[n][n];

    for (int i = 0; i < n; i++) { //let's make fist matrix
        for (int j = 0; j < n; j++) {
            psa[i][j] = (int)rand() % 100 + 1;
            psb[i][j] = (int)rand() % 100 + 1;
            c[i][j] = 0;
        }
    }

    for (int i = 0; i < n; i++) { //an the 2nd one
        for (int j = 0; j < n; j++) {
            pra[i][j] = psa[i][j];
            prb[i][j] = psb[i][j];

        }
    }
    MPI_Status statRecv[2];
    MPI_Request reqSend[2], reqRecv[2];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &proces);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);


    int PP = numprocs;
    double np = numprocs;
    kk = sqrt(np);
    int k = (int)kk;
    if (proces < k) // below neighbour set
    {
        left_neigh = (proces + k - 1) % k;
        right_neigh = (proces + k + 1) % k;
        up_neigh = ((k - 1) * k) + proces;
    }
    if (proces == k)
    {
        left_neigh = ((proces + k - 1) % k) + k;
        right_neigh = ((proces + k + 1) % k) + k;
        up_neigh = proces - k;
    }
```

```
    if (proces > k)
    {
        x = proces / k;
        left_neigh = ((proces + k - 1) % k) + x * k;
        right_neigh = ((proces + k + 1) % k) + x * k;
        up_neigh = proces - k;
    }
    if (proces == 0 || (proces / k) < (k - 1))
    {
        down_neigh = proces + k;
    }
    if ((proces / k) == (k - 1))
    {
        down_neigh = proces - ((k - 1) * k);
    }
    x = 0;

    for (int kk = 0; kk < PP; kk++) //algorithm
    {
        for (int i = 0; i < n / PP; i++)
        {
            for (int j = 0; j < n / PP; j++)
            {
                for (int k = 0; k < n / PP; k++)
                {
                    c[i][j] += psa[i][k] * psb[k][j];
                }
            }
        }
        MPI_Irecv(pra, n * n / PP / PP, MPI_FLOAT, left_neigh, tag, MPI_COMM_WORLD,
reqRecv);
        MPI_Irecv(prb, n * n / PP / PP, MPI_FLOAT, down_neigh, tag, MPI_COMM_WORLD,
&reqRecv[1]);
        MPI_Isend(psa, n * n / PP / PP, MPI_FLOAT, right_neigh, tag, MPI_COMM_WORLD,
reqSend);
        MPI_Isend(psb, n * n / PP / PP, MPI_FLOAT, up_neigh, tag, MPI_COMM_WORLD,
&reqSend[1]);
        MPI_Wait(reqRecv, statRecv);

    }

    cout << "A" << endl; //show result
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << pra[i][j] << " ";
        }
        cout << endl;
    }
    cout << "B" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << prb[i][j] << " ";
        }
        cout << endl;
    }
    cout << "C" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
```

```
            cout << c[i][j] << " ";
        }
        cout << endl;
    }


    MPI_Finalize();

    return 0;
}
```

Bài 3: Nhân ma trận với thuật toán Fox với MPI

- Nghiên cứu thuật toán nhân ma trận Fox
- Lập trình song song với MPI
- So sánh thời gian chạy song song với trường hợp chạy tuần tự, tính hệ số speedup, hệ số hiệu quả và só sánh với kết quả tính theo lý thuyết, giải thích.
- Đối sánh lại mã chương trình với thuật toán nhân ma trận Cannon.

//code of Fox' algorithm

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

#define N 9 /* dimension of the input matrix */

int matrixA[N][N];
int matrixB[N][N];
typedef struct {
      int p; /* number of processors */
      MPI_Comm comm; /* handle to global grid communicator */
      MPI_Comm row_comm; /* row communicator */
      MPI_Comm col_comm; /* column communicator */
      int q; /* dimension of the grid, = sqrt(p) */
      int my_row; /* row position of a processor in a grid */
      int my_col; /* column position of a procesor in a grid */
      int my_rank; /* rank within the grid */
}GridInfo;


void SetupGrid(GridInfo* grid)
{
      int old_rank;
      int dimensions[2];
      int wrap_around[2];
      int coordinates[2];
      int free_coords[2];

      /* get the overall information before overlaying cart_grid */

      MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
      MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
```

```
	/* Assumption: p is a perfect square */
	grid->q = (int)sqrt((double)grid->p);
	/* set the dimensions */
	dimensions[0] = dimensions[1] = grid->q;

	/* we want a torus on the second dimension, so set it appropriately */

	wrap_around[0] = 0;
	wrap_around[1] = 1;

	MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, wrap_around, 1, &(grid-
>comm));
	/* since we have set reorder to true, this might have changed the ranks */
	MPI_Comm_rank(grid->comm, &(grid->my_rank));
	/* get the cartesian coordinates for the current process */
	MPI_Cart_coords(grid->comm, grid->my_rank, 2, coordinates);
	/* set the coordinate values for the current coordinate */
	grid->my_row = coordinates[0];
	grid->my_col = coordinates[1];

	/* create row communicators */
	free_coords[0] = 0;
	free_coords[1] = 1; /* row is gonna vary */
	MPI_Cart_sub(grid->comm, free_coords, &(grid->row_comm));

	/* create column communicators */
	free_coords[0] = 1;
	free_coords[1] = 0; /* row is gonna vary */
	MPI_Cart_sub(grid->comm, free_coords, &(grid->col_comm));

}

/* normal matrix multiplication stuff */

void matmul(int** a, int** b, int** c, int size)
{
	int i, j, k;

	int** temp = (int**)malloc(size * sizeof(int*));
	for (i = 0; i < size; i++)
		*(temp + i) = (int*)malloc(size * sizeof(int));

	for (i = 0; i < size; i++)
	{
		for (j = 0; j < size; j++)
		{
			temp[i][j] = 0;
			for (k = 0; k < size; k++) {
				temp[i][j] = temp[i][j] + (a[i][k] * b[k][j]);
			}
		}
	}

	for (i = 0; i < size; i++)
		for (j = 0; j < size; j++)
			c[i][j] += temp[i][j];

}
```

```
void transfer_data_from_buff(int* buff, int** a, int buffsize, int row, int col) {

	if (buffsize != row * col)
	{
		printf("transfer_data_from_buf: buffer size does not match matrix
size!\n");
		exit(1);
	}
	int count = 0, i, j;
	for (i = 0; i < row; i++)
	{
		for (j = 0; j < col; j++) {
			a[i][j] = buff[count];
			count++;
		}
	}
}

void transfer_data_to_buff(int* buff, int** a, int buffsize, int row, int col) {

	if (buffsize != row * col)
	{
		printf("transfer_data_to_buf: buffer size does not match matrix
size!");
		exit(1);
	}
	int count = 0, i, j;
	for (i = 0; i < row; i++)
	{
		for (j = 0; j < col; j++) {
			buff[count] = a[i][j];
			count++;
		}
	}
}

void Fox(int n, GridInfo* grid, int** a, int** b, int** c)
{
	int** tempa;
	int* buff; /* buffer for Bcast & send_recv */
	int stage;
	int root;
	int submat_dim; /* = n/q */
	int source;
	int dest;
	int i;
	MPI_Status status;

	submat_dim = n / grid->q;

	/* Initialize tempa */
	tempa = (int**)malloc(submat_dim * sizeof(int*));
	for (i = 0; i < submat_dim; i++)
		*(tempa + i) = (int*)malloc(submat_dim * sizeof(int));
	/* initialize buffer */
	buff = (int*)malloc(submat_dim * submat_dim * sizeof(int));
```

```
	/* we are gonna shift the elements of matrix b upwards with the column fixed
*/
	source = (grid->my_row + 1) % grid->q; /* pick the emmediately lower element
*/
	dest = (grid->my_row + grid->q - 1) % grid->q; /* move current element to
immediately upper row */


	for (stage = 0; stage < grid->q; stage++)
	{
		root = (grid->my_col + stage) % grid->q;
		if (root == grid->my_col)
		{
			transfer_data_to_buff(buff, a, submat_dim * submat_dim,
submat_dim, submat_dim);
			MPI_Bcast(buff, submat_dim * submat_dim, MPI_INT, root, grid-
>row_comm);
			transfer_data_from_buff(buff, a, submat_dim * submat_dim,
submat_dim, submat_dim);

			matmul(a, b, c, submat_dim);
		}
		else
		{
			transfer_data_to_buff(buff, tempa, submat_dim * submat_dim,
submat_dim, submat_dim);
			MPI_Bcast(buff, submat_dim * submat_dim, MPI_INT, root, grid-
>row_comm);
			transfer_data_from_buff(buff, tempa, submat_dim * submat_dim,
submat_dim, submat_dim);

			matmul(tempa, b, c, submat_dim);
		}
		transfer_data_to_buff(buff, b, submat_dim * submat_dim, submat_dim,
submat_dim);
		MPI_Sendrecv_replace(buff, submat_dim * submat_dim, MPI_INT, dest, 0,
source, 0, grid->col_comm, &status);
		transfer_data_from_buff(buff, b, submat_dim * submat_dim, submat_dim,
submat_dim);
	}

}

void initialiseAB()
{
	int i, j;
	/*
**************************************************************************
*****
	Initialize the input matrix
	Note: This initalization is deterministic & hence is done by every process in
the same way
		I wanted to design a fully distributed program, hence I took this
strategy.
		A better strategy could have been to let master alone initialize the
matrices & then
		send the slaves their local copy only
```

```
	*********************************************************************
**************/
	for (i = 0; i < N; i++)
	{
		for (j = 0; j < N; j++)
		{
			matrixA[i][j] = i + j;
			matrixB[i][j] = i + j;
		}
	}
}


int main(int argc, char* argv[])
{

	int i, j, dim;
	int** localA;
	int** localB;
	int** localC;
	MPI_Init(&argc, &argv);

	GridInfo grid;
	/*initialize Grid */

	SetupGrid(&grid);
	/* Initialize matrix A & B */
	initialiseAB();
	/* calculate local matrix dimension */
	dim = N / grid.q;
	/* allocate space for the three matrices */


	localA = (int**)malloc(dim * sizeof(int*));

	localB = (int**)malloc(dim * sizeof(int*));

	localC = (int**)malloc(dim * sizeof(int*));

	for (i = 0; i < dim; i++)
	{
		*(localA + i) = (int*)malloc(dim * sizeof(int));
		*(localB + i) = (int*)malloc(dim * sizeof(int));
		*(localC + i) = (int*)malloc(dim * sizeof(int));
	}


	/* Compute local matrices - Ideally the master should do this & pass it onto
all the slaves */
	/* At the same time initialize localC to all zeros */

	int base_row = grid.my_row * dim;
	int base_col = grid.my_col * dim;

	for (i = base_row; i < base_row + dim; i++)
	{
		for (j = base_col; j < base_col + dim; j++)
		{
```

```
                        localA[i - (base_row)][j - (base_col)] = matrixA[i][j];
                        localB[i - (base_row)][j - (base_col)] = matrixB[i][j];
                        localC[i - (base_row)][j - (base_col)] = 0;
                }
        }



        Fox(N, &grid, localA, localB, localC);

        /* print results */
        printf("rank=%d, row=%d col=%d\n", grid.my_rank, grid.my_row, grid.my_col);
        for (i = 0; i < dim; i++)
        {
                for (j = 0; j < dim; j++)
                {
                        //printf("localC[%d][%d]=%d ", i,j,localC[i][j]);
                        printf("%d ", localC[i][j]);
                }
                printf("\n");
        }
        MPI_Finalize();
        exit(0);

}
```

Bài 4: Nhân ma trận với cấu trúc lai MPI/OpenMP

- **Chạy chuong trình với lệnh:**
  mpiexec -n 4 <tên chương trình exe> <N> < numthread>
  với N là kích cỡ ma trận.

```
//code
#include <omp.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define TAG 13

int main(int argc, char* argv[]) {
    double** A, ** B, ** C, * tmp;
    double startTime, endTime;
    int numElements, offset, stripSize, myrank, numnodes, N, i, j, k;
    int numThreads, chunkSize = 10;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numnodes);
    N = atoi(argv[1]);
    numThreads = atoi(argv[2]);  // difference from MPI: how many threads/rank?
     omp_set_num_threads(numThreads);  // OpenMP call to set threads per rank
    // allocate A, B, and C --- note that you want these to be
    // contiguously allocated.  Workers need less memory allocated.
    if (myrank == 0) {
        tmp = (double*)malloc(sizeof(double) * N * N);
        A = (double**)malloc(sizeof(double*) * N);
        for (i = 0; i < N; i++)
```

```
            A[i] = &tmp[i * N];
    }
    else {
        tmp = (double*)malloc(sizeof(double) * N * N / numnodes);
        A = (double**)malloc(sizeof(double*) * N / numnodes);
        for (i = 0; i < N / numnodes; i++)
            A[i] = &tmp[i * N];
    }
    tmp = (double*)malloc(sizeof(double) * N * N);
    B = (double**)malloc(sizeof(double*) * N);
    for (i = 0; i < N; i++)
        B[i] = &tmp[i * N];
    if (myrank == 0) {
        tmp = (double*)malloc(sizeof(double) * N * N);
        C = (double**)malloc(sizeof(double*) * N);
        for (i = 0; i < N; i++)
            C[i] = &tmp[i * N];
    }
    else {
        tmp = (double*)malloc(sizeof(double) * N * N / numnodes);
        C = (double**)malloc(sizeof(double*) * N / numnodes);
        for (i = 0; i < N / numnodes; i++)
            C[i] = &tmp[i * N];
    }

    if (myrank == 0) {
        // initialize A and B
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                A[i][j] = 1.0;
                B[i][j] = 1.0;
            }
        }
    }
    // start timer
    if (myrank == 0) {
        startTime = MPI_Wtime();
    }
    stripSize = N / numnodes;

    // send each node its piece of A -- note could be done via MPI_Scatter
    if (myrank == 0) {
        offset = stripSize;
        numElements = stripSize * N;
        for (i = 1; i < numnodes; i++) {
            MPI_Send(A[offset], numElements, MPI_DOUBLE, i, TAG, MPI_COMM_WORLD);
            offset += stripSize;
        }
    }
    else {  // receive my part of A
        MPI_Recv(A[0], stripSize * N, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }
    // everyone gets B
    MPI_Bcast(B[0], N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Let each process initialize C to zero
    for (i = 0; i < stripSize; i++) {
```

```
        for (j = 0; j < N; j++) {
            C[i][j] = 0.0;
        }
    }
    // do the work---this is the primary difference from the pure MPI program
#pragma omp parallel for shared(A,B,C,numThreads) private(i,j,k) schedule (static,
chunkSize)
    for (i = 0; i < stripSize; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    // master receives from workers  -- note could be done via MPI_Gather
    if (myrank == 0) {
        offset = stripSize;
        numElements = stripSize * N;
        for (i = 1; i < numnodes; i++) {
            MPI_Recv(C[offset], numElements, MPI_DOUBLE, i, TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            offset += stripSize;
        }
    }
    else { // send my contribution to C
        MPI_Send(C[0], stripSize * N, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
    }
    // stop timer
    if (myrank == 0) {
        endTime = MPI_Wtime();
        printf("Time is %f\n", endTime - startTime);
    }
    // print out matrix here, if I'm the master
    if (myrank == 0 && N < 10) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                printf("%f ", C[i][j]);
            }
            printf("\n");
        }
    }
    MPI_Finalize();
    return 0;
}
```

----------HẾT----------