```c
/*
 * Parallelizing 2D Heat Equations solver using 5 points equations
 *
 * Author: Dustin (Ting-Hsuan) Ma
 *
 * To Compile: mpicc -o MPI.exe -lm mpiHeat.c
 * To Run: mpirun -np 4 ./MPI.exe
 *
 */

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

typedef double       REAL;
typedef const double C_REAL;
typedef int          INT;

// Spacial
#define LX ( REAL ) 20.0f
#define LY ( REAL ) 20.0f
#define NX ( INT ) 10
#define NY ( INT ) 10
#define DX LX / (( REAL ) NX - 1.0f)
#define DY LY / (( REAL ) NY - 1.0f)

// Temperature
#define TMAX ( REAL ) 100.0f
#define TMIN ( REAL ) 0.0f

// Time
#define DT ( REAL ) 0.25f * DY *DY
#define MAXITER 500

// Calculation index
#define IC i + j *NX
#define IP1 (i + 1) + j *NX
#define IM1 (i - 1) + j *NX
#define JP1 i + (j + 1) * NX
#define JM1 i + (j - 1) * NX

// Process
#define MASTER 0

void initializeM(REAL *in, const int nrow, const int nGhostLayers)
{
    for (INT j = 0; j < nrow + nGhostLayers; j++) {
        for (INT i = 0; i < NX; i++) {
            if (j == 1) {
                in[ IC ] = TMAX;
            }
            if (j == nrow + 1) {
                in[ IC ] = TMIN;
            }
            if (i == 0) {
                in[ IC ] = TMIN;
            }
            if (i == NX - 1) {
                in[ IC ] = TMIN;
            }
        }
    }
}

void decomposeMesh_1D(const int N, const int nProcs, const int myRank, int *start, int *end)
{
    *start = myRank * N / nProcs;
    *end   = *start + N / nProcs;
}

void SolveHeatEQ(C_REAL *now, REAL *out, const int nrow, const int myRank, const int nProcs)
{
    int startLoc, endLoc;

    /* Special condition that applies for ROOT process */
    if (myRank == MASTER)
        startLoc = 2;
    else
        startLoc = 1;

    /* Special condition that applied for nProcs-1 process */
    if (myRank == nProcs - 1)
        endLoc = nrow;
    else
        endLoc = nrow + 1;
```



```c
    /* Heat solving iteration only performs calculations on the center cells of matrix */
    for (INT j = startLoc; j < endLoc; j++) {
        for (INT i = 1; i < NX - 1; i++) {
            out[ IC ] = (((now[ IP1 ] - 2.0f * now[ IC ] + now[ IM1 ]) / (DX * DX))
                         + ((now[ JP1 ] - 2.0f * now[ IC ] + now[ JM1 ])) / (DY * DY))
                        * DT
                        + now[ IC ];
        }
    }
}

/* Blocking send and receive
 *
 * Sperately sends NX values down and up, and the respective
 * process initiate a receive call.
 */
void exchange_Send_and_Recieve(REAL *in, const int src, const int dest, const int nrow,
                               const int myRank)
{
    int tag0 = 0;                                                                  // up send tag
    int tag1 = 1;                                                                  // down send tag
    MPI_Send(in + (nrow * NX), NX, MPI_DOUBLE, dest, tag1, MPI_COMM_WORLD); // send to down
    MPI_Send(in + NX, NX, MPI_DOUBLE, src, tag0, MPI_COMM_WORLD);           // send to up

    MPI_Recv(in, NX, MPI_DOUBLE, src, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // receive from up
    MPI_Recv(in + ((nrow + 1) * NX), NX, MPI_DOUBLE, dest, tag0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE); // receive from down
}

/* Nonblocking send&receive
 *
 * The function sends NX values to the bottom process.
 * In return, the bottom process also sends NX values
 * back up to the top process
 */
void exchange_SendRecv(REAL *in, const int src, const int dest, const int nrow, const int myRank)
{
    int tag0 = 0; // send tag
    int tag1 = 1; // send tag
    MPI_Sendrecv(in + (nrow * NX), NX, MPI_DOUBLE, dest, tag0, in, NX, MPI_DOUBLE, src, tag0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Sending down
    MPI_Sendrecv(in + NX, NX, MPI_DOUBLE, src, tag1, in + (nrow + 1) * NX, NX, MPI_DOUBLE, dest,
                 tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Sending up
}

void outputMatrix(C_REAL *in)
{
    for (INT j = 0; j < NY; j++) {
        for (INT i = 0; i < NX; i++) {
            printf("%8.4f", in[ IC ]);
        }
        printf("\n");
    }
    printf("\n");
}

void print2Display(C_REAL *in, const int start, const int end, const int nrow, const int myRank,
                   const int nProcs)
{
    int nGhostLayers = 2;
    for (int j = 0; j < nrow + nGhostLayers; j++) {
        for (int i = 0; i < NX; i++) {
            printf("%8.4f", in[ IC ]);
        }
        printf("\n");
    }
}

int main(int argc, char **argv)
{
    int nProcs; // number of processes
    int myRank; // process rank
    int src;    // handles for communication, source process id
    int dest;   // handles for communication, destination process id
    int start;  // start index for each partial domain
    int end;    // end index for each partial domain
    int nrow;   // number of row needed to be allocated by local array

    MPI_Init(&argc, &argv);                 // initialize MPI
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs); // get the number of processes

    int nDims = 1; // dimension of Cartesian decomposition 1 => slices
    int dimension[ nDims ];
    int isPeriodic[ nDims ];
    int reorder = 1; // allow system to optimize(reorder) the mapping of processes to physical cores
```

```
    dimension[ 0 ]  = nProcs;
    isPeriodic[ 0 ] = 0; // periodicty of each dimension

    MPI_Comm comm1D; // define a communicator that would be assigned a new topology
    MPI_Cart_create(MPI_COMM_WORLD, nDims, dimension, isPeriodic, reorder, &comm1D);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get the rank of a process after REORDERING!
    MPI_Cart_shift(comm1D, 0, 1, &src,
                   &dest); // Let MPI find out the rank of processes for source and destination

    // Mesh Decompotistion
    decomposeMesh_1D(NY, nProcs, myRank, &start, &end);
    nrow = (end - start);
    printf("myRank=%d, mySource=%2.1d, myDestination=%2.1d, nrow=%2.1d, start=%2.1d, end=%2.1d\n",
           myRank, src, dest, nrow, start, end);
    int nGhostLayers = 2;
    int AllocSize    = (nrow + nGhostLayers) * NX;

    // Allocating Memory for every process after mesh decomposition
    REAL *local, *local_new, *tmp, *theta_new;
    local     = ( REAL * ) calloc(AllocSize, sizeof(*local));
    local_new = ( REAL * ) calloc(AllocSize, sizeof(*local_new));

    // Allocating/Initializing only within the Root Process
    if (myRank == MASTER) {
        theta_new
        = ( REAL * ) calloc(NX * NY, sizeof(*theta_new)); // Final output memory allocation
        initializeM(local, nrow, nGhostLayers);
        initializeM(local_new, nrow, nGhostLayers);
    }

    // Performing calculation and timing for scalibility
    MPI_Barrier(MPI_COMM_WORLD);
    double startT = MPI_Wtime( );
    for (int iter = 0; iter < MAXITER; iter++) {
        // exchange_Send_and_Recieve(local, src, dest, nrow, myRank);   //Blocking
        exchange_SendRecv(local, src, dest, nrow, myRank); // Nonblocking
        SolveHeatEQ(local, local_new, nrow, myRank, nProcs);

        tmp       = local;
        local     = local_new;
        local_new = tmp;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    double finishT = MPI_Wtime( );

    // Gather data from rest of the processes into Root
    MPI_Gather(local + NX, nrow * NX, MPI_DOUBLE, theta_new, nrow * NX, MPI_DOUBLE, MASTER,
               MPI_COMM_WORLD);

    /*    // Testing
        if (myRank == MASTER) {
            printf("***********Debug Matrix***********\n");
            print2Display(local, start, end, nrow, myRank, nProcs);
        }
    */

    // Output data from Root
    if (myRank == MASTER) {
        printf("**********FINAL OUTPUT AFTER GATHER**********\n");
        outputMatrix(theta_new);
    }

    // Barrier before recording the finish time
    double elapsedTime = finishT - startT;
    double wallTime;
    MPI_Reduce(&elapsedTime, &wallTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    if (myRank == 0) {
        printf("Wall-clock time = %.3f (ms) \n", wallTime * 1e3);
    }

    // Deallocating Arrays
    free(local);
    free(local_new);
    free(theta_new);

    local     = NULL;
    local_new = NULL;
    theta_new = NULL;

    MPI_Finalize( );
    return EXIT_SUCCESS;
}
```