CMPE 300

ALGORITHM ANALYSIS

Instructor: Tunga Güngör

Student: Koray Çetin

MPI Project

Programming Project

Due Date: 20/12/2019 17:00

1. Introduction

Our focus will be on the particular cellular automaton called the (Conway's) Game of Life, devised by J. H. Conway in 1970. Just "Game" for short. In the Game, we have a 2-dimensional orthogonal grid as a map (i.e. a matrix). Each cell on the map can either contain a creature (1) or be empty (0).

The 8 cells that are immediately around a cell are considered as its neighbors. Then, the rules of the Game are as follows:

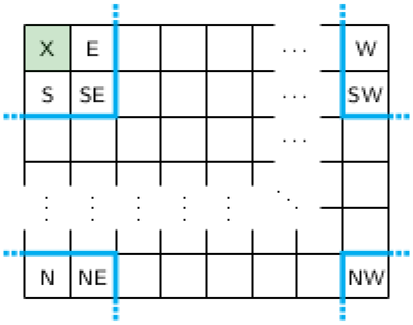
Loneliness kills: A creature dies (i.e. the cell becomes empty) if it has less than 2 neighboring creatures.

Overpopulation also kills: A creature dies (and becomes empty) if it has more than 3 neighboring creatures. See the following example. Note that the creature would not die if it had one less neighboring creature.

Reproduction: A new life appears on an empty cell if it has exactly 3 neighboring creatures. See the following example. Note that the creature would not be born if the cell had one more or one less neighboring creature.

In any other condition, the creatures remain alive, and the empty spaces remain empty.

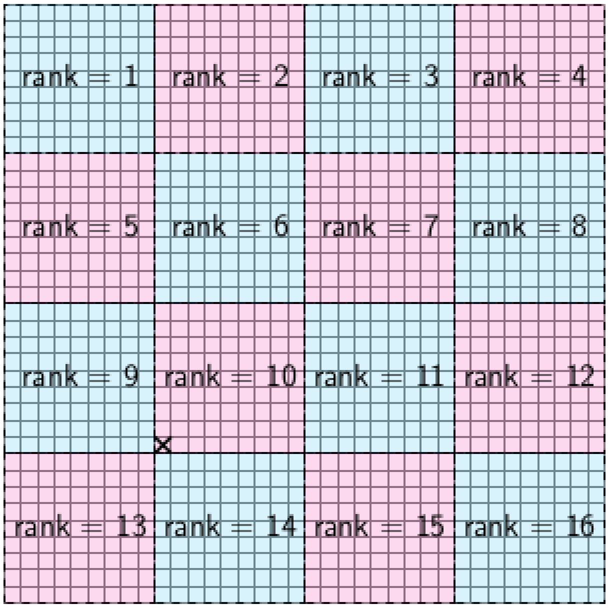
Boundaries: The neighbors are not missing! They are just at the opposite end of the map. See the following image for the neighborhood (with the blue "square") of the top left corner cell X. Letters N, S, E, and W denote the neighbors of X at North, South, East, and West. For instance, see how the "northern" neighbor (N) of X is at the bottom of the 2D array.



Program must be implemented in a parallel manner. Map of the game will be divided into subprocesses and the rules of the game will be implemented with communication between processes.

Splits: The maps that we will be using will always be 360 × 360. We will be using smaller square maps in this section, only to have visualizations that look better.

Let's assume a S x S map. The map will be split into C square arrays, each with S / √C rows and columns. The following image shows how this split would be with S = 36 and C = 16.



Solution: The main process reads the input and divides the map to other processes. At each iteration, processes send the necessary data to other processes. After that, each process updates it’s map by receiving information from other processes if it’s necessary. When desired number of iterations is reached, the main process recollects the whole map from other processes and writes the result to an output file.

2. Program Interface

To run this program, mpi environment must be installed. Here’s a link for downloading it:

https://www.open-mpi.org/doc/current/

To compile the program, user should open a command line and write the following command:

```
mpic++ game.cpp -o game
```

A compiled MPI program game, can be run with the following:

```
mpirun -np [M] --oversubscribe ./game input.txt output.txt [T]
```

[M] is the number of processes to run game on. If you want to have C = 8 worker processes, then you need 9 processes in total, accounting for the manager. Hence, you should write -np 9 in the command line. (M – 1) must be a perfect square and a divisor of 360. The flag --oversubscribe allows you to set [M] more than just the number of logical cores on your machine, which we will do. Arguments input.txt, output.txt, and [T] are passed onto game as command line arguments. The [T] is the number of iterations to simulate the Game. The output.txt should be filled with the map’s final state after [T] iterations of simulation, with the same syntax as in the input file, described above.

3. Input and Output

The input.txt will contain the initial state of the map of size 360 × 360 with;

- rows separated by a single new line (\n) character,

• each cell on a row separated by a single space ( ) character,

• each cell as a 0 or 1 for emptiness and life.

Here is an example input file of a 4 x 4 map:

```
1 1 0 1
1 0 1 0
0 0 1 1
0 1 1 1
```

If argument T is passed as 1, the program will perform 1 iteration and the output.txt file will be as follows:

```
0 1 0 0
1 0 0 0
0 0 0 0
0 0 0 0
```

To give an example, map size is minimized; but the program works with larger inputs.

4. Program Structure

4.1 divideGame

After MPI is initialized, main process reads the input file and divides the map as checkers to the other processes. It divides the game sequentially and incrementally to each process with MPI_Send function. It also sends the boundaries to them. Meanwhile, other processes are waiting the information with MPI_Recv function.

Each cell is passed to the processes with different tag and each subprocess is waiting the data with this tag parameter from the rank 0 process. Subprocesses wait for the cell values in given boundaries. They also initialize ups, downs, lefts, rights and corners vectors with –1 value. Later, these vectors will be used to store data from adjacent processes.

Each process has two game map arrays. This is for calculating a new state of the map while not distorting the old one. In each state, old map and new map changes. This is implemented by incrementing a state variable in each iteration, and selecting the map with mode calculation.

4.2 sendAll

In this function, each process besides the rank 0 process sends the necessary information to its neighbors. Each process gets the neighbor process's rank in specific direction with **getWorldRank** function. As a tag, it sends the index of the cell in related boundary. It also passes tag 0 to its diagonal neighbors.

4.3 getWorldRank

Gets a direction string and returns the world rank at that direction. It involves a lot of checkered map calculations and edge case comparisons.

4.3 updateAll

In this function, each process besides the rank 0 process iterates through its cells and calls the **update** function with specific indexes. **Update** function calculates the number of living neighbors of the cell and updates it.

4.4 update

This function gets the number of living neighbors of the cell in given coordinate with **nOfLivingNeighbors** function. Then selects the old and new game maps according to the state variable. New game map is updated in given cell with living neighbor count and game rules.

4.5 nOfLivingNeighbors

This function calculates each neighbor of a cell and then returns the sum of these values. If the cell is in the boundaries, calls the **getFromArrayOrReceive** function with a direction and an index; otherwise gets the neighbor's value from its own game map.

4.6 getFromArrayOrReceive

This function gets the place of the neighbor cell in its boundary vectors with **getPlace** function first. If the direction is diagonal, set the index to 0 for tag logic. Then checks the place's value, if its value is not –1, returns this value. Because this means, value needed is already received from the neighbor process. Otherwise, gets the world rank in given direction with **getWorldRank** function, receives the value from the process with that rank, and updates its boundary vector with this value.

4.7 getPlace

Gets and index and a direction, returns a pointer to process' boundary vector's element in that index and direction.

4.8 Iterations

In a for loop **sendAll** and **updateAll** functions are called repeatedly. The time when a process enters the **updateAll** function, it sends all the information needed to other processes in **sendAll** function. This method eliminates the deadlock possibility. Also, these methods choose the old game map and new game map by looking to the stage parameter.

4.9 recollect

In this function, all processes except the rank zero process sends their maps to the rank zero process cell by cell. Meanwhile, rank zero process receives the data from other processes and puts them into its own map.

5. Examples

Let's examine a 4 x 4 matrix. And the matrix after one iteration.

$$\begin{matrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{matrix} \quad -> \quad \begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Number of living neighbors = NW + N + NE + E + W + SW + S + SE

For cell (0,0), there are 1 + 1 + 0 + 1 + 1 + 0 + 0 + 0 = 4 living neighbors. So, overpopulation kills the cell.

For cell (0,1), there are 1 + 0 + 0 + 1 + 0 + 0 + 0 + 1 = 3 living neighbors. So, neighbor cells reproduce.

For cell (0,2), there are 0 + 0 +1 + 1 + 1 + 0 + 1 + 0 = 4 living neighbors. So, overpopulation kills the cell.

For cell (0,3), there are 0 + 1 + 1 + 0 + 1 + 1 + 0 + 0 = 4 living neighbors. So, overpopulation kills the cell.

For cell (1,0), there are 1 + 1 + 1 + 0 + 0 + 0 + 0 + 0 = 3 living neighbors. So, neighbor cells reproduce.

For cell (1,1), there are 1 + 1 + 0 + 0 + 1 + 0 + 0 + 0 = 3 living neighbors. So, neighbor cells reproduce.

For cell (1,2), there are 1 + 0 + 1 + 0 + 0 + 0 + 0 + 0 = 2 living neighbors. So, the cell remains unchanged.

For cell (1,3), there are 0 + 1 + 1 + 1 + 0 + 0 + 0 + 0 = 3 living neighbors. So, neighbor cells reproduce.

For cell (2,0), there are 0 + 0 + 0 + 0 + 0 + 1 + 1 + 0 = 2 living neighbors. So, the cell remains unchanged.

For cell (2,1), there are 0 + 0 + 1 + 0 + 0 + 1 + 0 + 0 = 2 living neighbors. So, the cell remains unchanged.

For cell (2,2), there are 1 + 0 + 1 + 0 + 0 + 0 + 0 + 0 = 2 living neighbors. So, the cell remains unchanged.

For cell (2,3), there are 1 + 0 + 0 + 0 + 0 + 0 + 1 + 1 = 3 living neighbors. So, neighbor cells reproduce.

For cell (3,0), there are 0 + 0 + 0 + 1 + 0 + 1 + 1 + 0 = 3 living neighbors. So, neighbor cells reproduce.

For cell (3,1), there are 0 + 0 + 0 + 1 + 0 + 1 + 1 + 0 = 3 living neighbors. So, neighbor cells reproduce.

For cell (3,2), there are 0 + 0 + 0 + 0 + 1 + 1 + 0 + 1 = 3 living neighbors. So, neighbor cells reproduce.

For cell (3,3), there are 0 + 0 + 0 + 0 + 1 + 0 + 1 + 1 = 3 living neighbors. So, neighbor cells reproduce.

6. Improvements and Extensions

- **divideGame** and **recollect** functions sends and receives the cells from other processes one by one. Instead these functions may send and receive arrays. Since they are called only once in each execution, the functions are implemented in this way.
- Dividing the send and receive parts completely in **sendAll** and **updateAll** functions is really a safe way. However, one process can send the current cell value immediately after it calculates it. This approach could cause data misinterpretations, but by using a different tag in each iteration, these risks may be prevented.

7. Difficulties Encountered

C++ isn't a language I use every day and it's not very user-friendly. In this situation, some vector functions may be hard to use. When I was dividing and recollecting the game from rank zero process, I first tried to send and receive cells as blocks. But I couldn't fix segmentation fault error. Hence, I decided to send and receive the cells one by one.

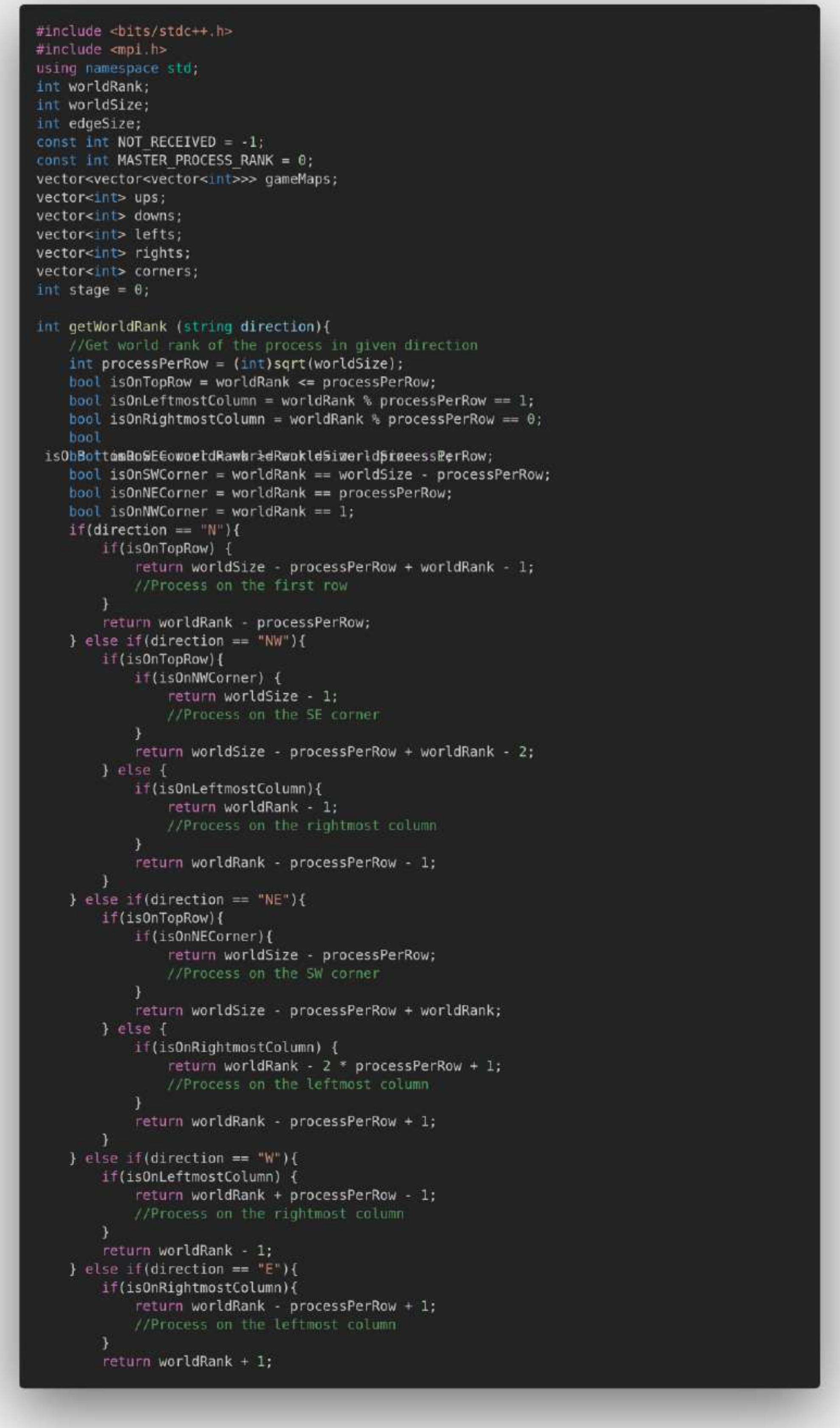
Calculating neighbors was hard because of the checkered structure. For making the program to work properly, there are lot of edge cases to calculate with mod and division operations.

I also tried to send information immediately after calculating the new cell. But couldn't manage the solve deadlocks.

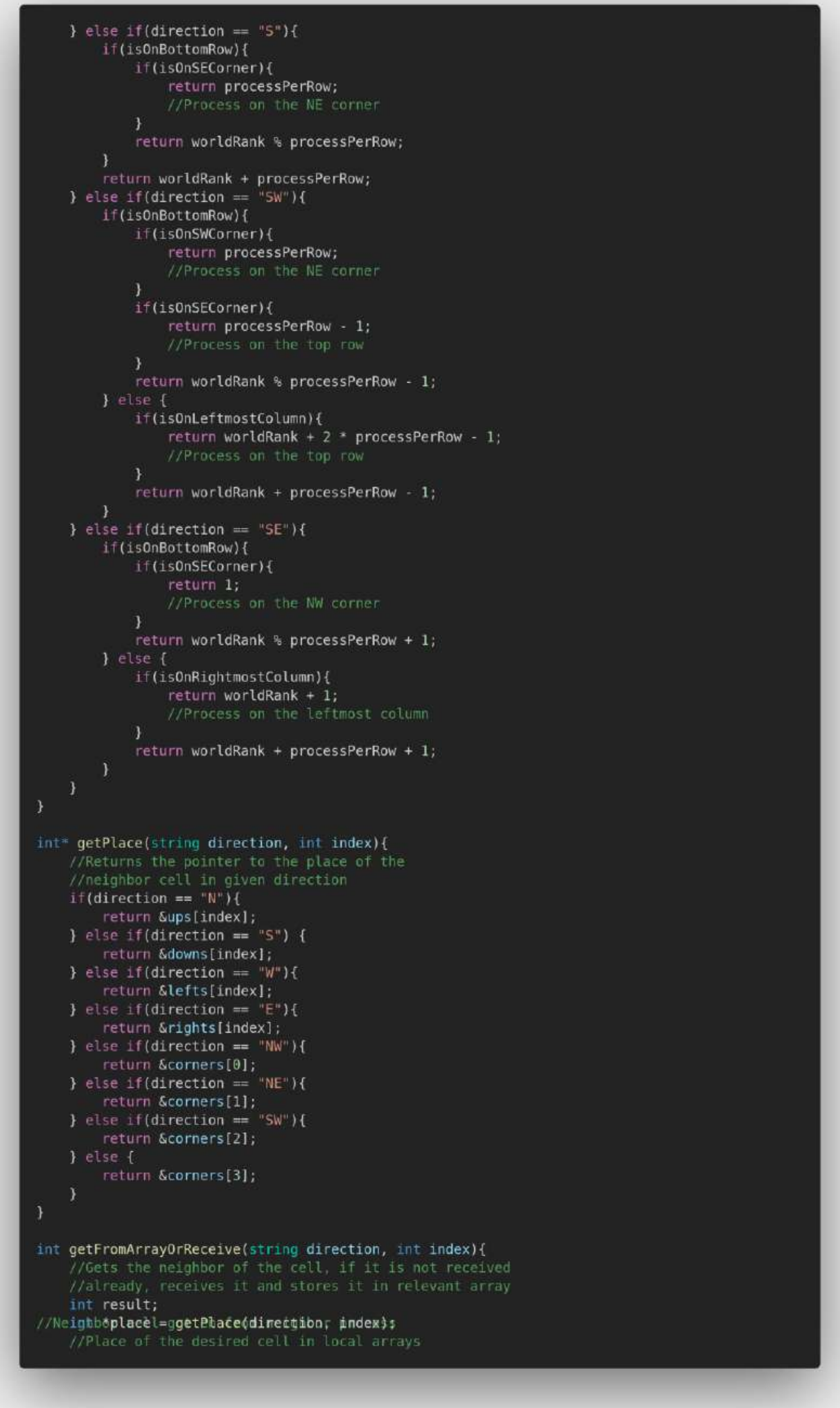
8. Conclusion

Project's game logic was not that hard to implement. But splitting the execution into processes and make them communicate with each other was hard. It was a great parallel programming practice and was helpful to understand the concepts of it. Also, project was involved a lot of calculations due to map splitting logic.

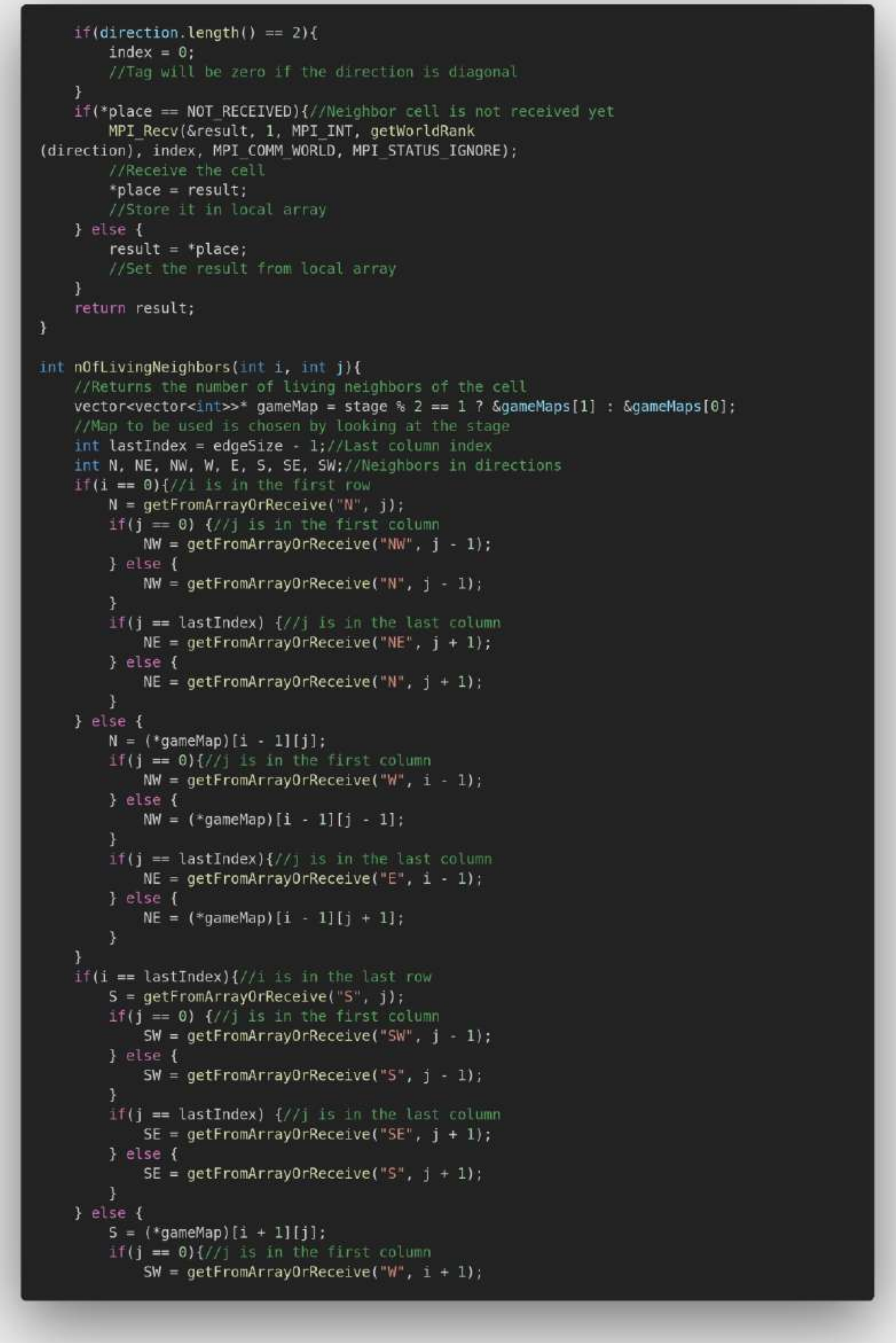
9. Appendix



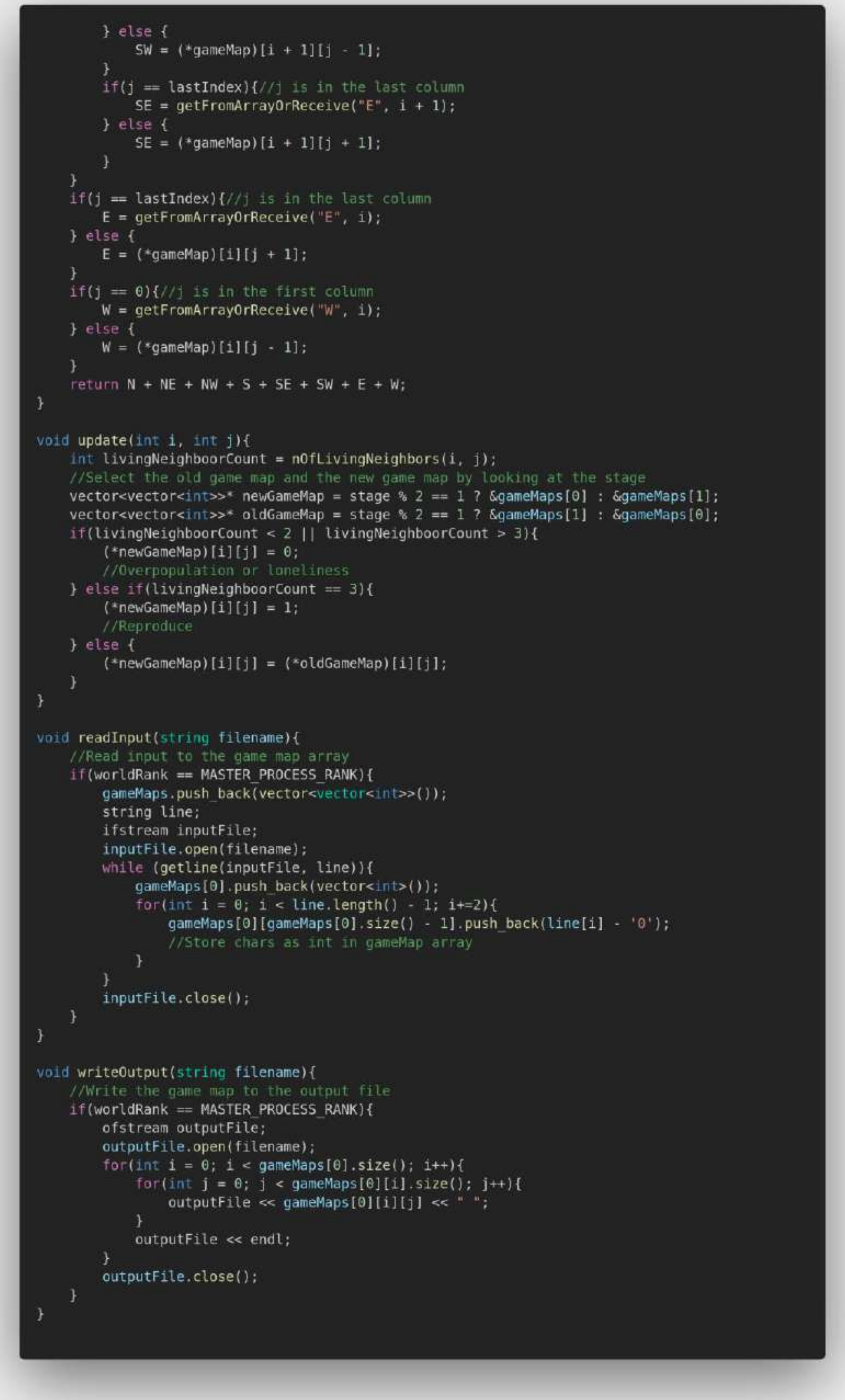
```cpp
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;
int worldRank;
int worldSize;
int edgeSize;
const int NOT_RECEIVED = -1;
const int MASTER_PROCESS_RANK = 0;
vector<vector<vector<int>>> gameMaps;
vector<int> ups;
vector<int> downs;
vector<int> lefts;
vector<int> rights;
vector<int> corners;
int stage = 0;

int getWorldRank (string direction){
    //Get world rank of the process in given direction
    int processPerRow = (int)sqrt(worldSize);
    bool isOnTopRow = worldRank <= processPerRow;
    bool isOnLeftmostColumn = worldRank % processPerRow == 1;
    bool isOnRightmostColumn = worldRank % processPerRow == 0;
    bool
isOnBottomRow = worldRank > worldSize - processPerRow;
    bool isOnSECorner = worldRank == worldSize;
    bool isOnSWCorner = worldRank == worldSize - processPerRow;
    bool isOnNECorner = worldRank == processPerRow;
    bool isOnNWCorner = worldRank == 1;
    if(direction == "N"){
        if(isOnTopRow) {
            return worldSize - processPerRow + worldRank - 1;
            //Process on the first row
        }
        return worldRank - processPerRow;
    } else if(direction == "NW"){
        if(isOnTopRow){
            if(isOnNWCorner) {
                return worldSize - 1;
                //Process on the SE corner
            }
            return worldSize - processPerRow + worldRank - 2;
        } else {
            if(isOnLeftmostColumn){
                return worldRank - 1;
                //Process on the rightmost column
            }
            return worldRank - processPerRow - 1;
        }
    } else if(direction == "NE"){
        if(isOnTopRow){
            if(isOnNECorner){
                return worldSize - processPerRow;
                //Process on the SW corner
            }
            return worldSize - processPerRow + worldRank;
        } else {
            if(isOnRightmostColumn) {
                return worldRank - 2 * processPerRow + 1;
                //Process on the leftmost column
            }
            return worldRank - processPerRow + 1;
        }
    } else if(direction == "W"){
        if(isOnLeftmostColumn) {
            return worldRank + processPerRow - 1;
            //Process on the rightmost column
        }
        return worldRank - 1;
    } else if(direction == "E"){
        if(isOnRightmostColumn){
            return worldRank - processPerRow + 1;
            //Process on the leftmost column
        }
        return worldRank + 1;
```



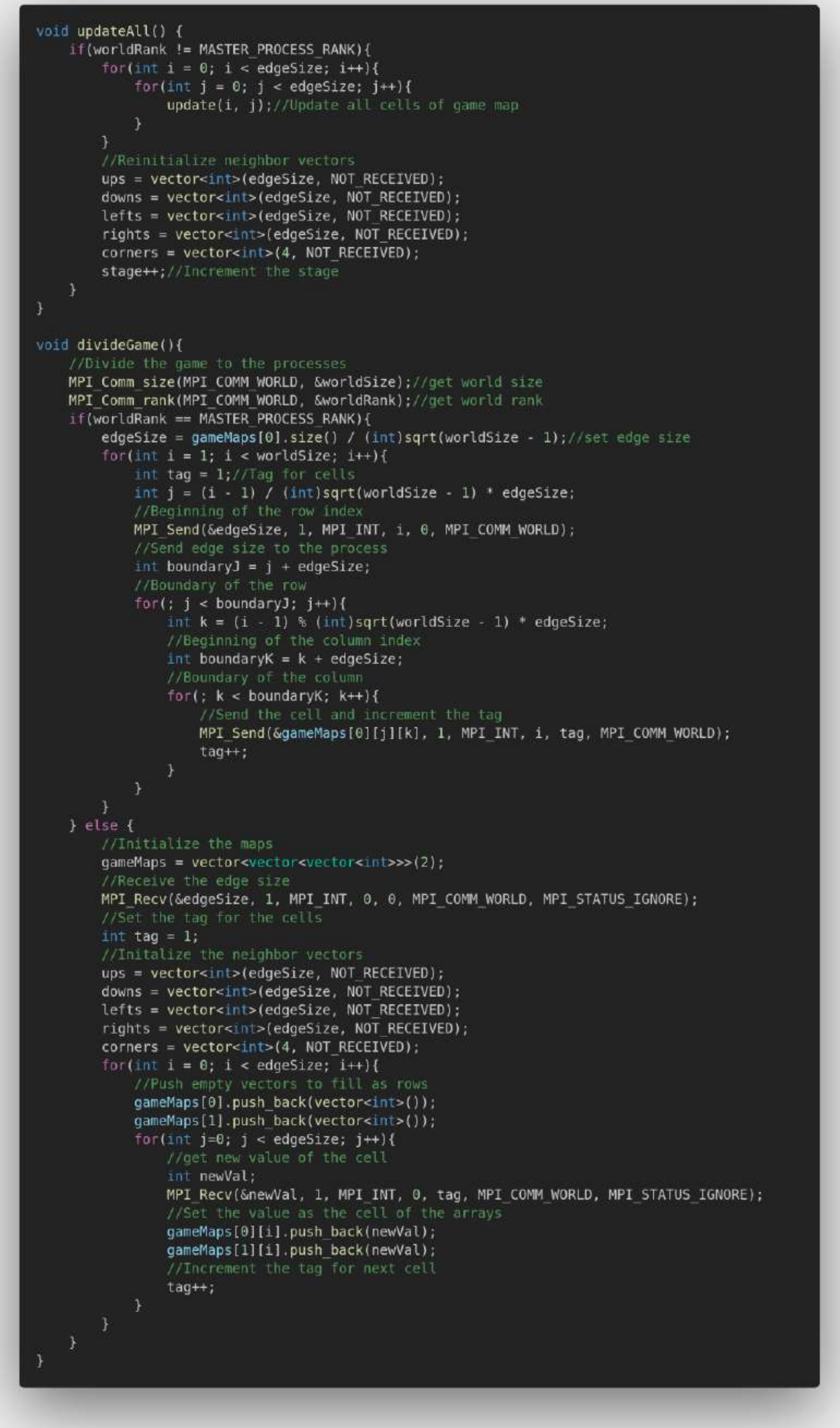
```
    } else if(direction == "S"){
        if(isOnBottomRow){
            if(isOnSECorner){
                return processPerRow;
                //Process on the NE corner
            }
            return worldRank % processPerRow;
        }
        return worldRank + processPerRow;
    } else if(direction == "SW"){
        if(isOnBottomRow){
            if(isOnSWCorner){
                return processPerRow;
                //Process on the NE corner
            }
            if(isOnSECorner){
                return processPerRow - 1;
                //Process on the top row
            }
            return worldRank % processPerRow - 1;
        } else {
            if(isOnLeftmostColumn){
                return worldRank + 2 * processPerRow - 1;
                //Process on the top row
            }
            return worldRank + processPerRow - 1;
        }
    } else if(direction == "SE"){
        if(isOnBottomRow){
            if(isOnSECorner){
                return 1;
                //Process on the NW corner
            }
            return worldRank % processPerRow + 1;
        } else {
            if(isOnRightmostColumn){
                return worldRank + 1;
                //Process on the leftmost column
            }
            return worldRank + processPerRow + 1;
        }
    }
}

int* getPlace(string direction, int index){
    //Returns the pointer to the place of the
    //neighbor cell in given direction
    if(direction == "N"){
        return &ups[index];
    } else if(direction == "S") {
        return &downs[index];
    } else if(direction == "W"){
        return &lefts[index];
    } else if(direction == "E"){
        return &rights[index];
    } else if(direction == "NW"){
        return &corners[0];
    } else if(direction == "NE"){
        return &corners[1];
    } else if(direction == "SW"){
        return &corners[2];
    } else {
        return &corners[3];
    }
}

int getFromArrayOrReceive(string direction, int index){
    //Gets the neighbor of the cell, if it is not received
    //already, receives it and stores it in relevant array
    int result;
    int* place = getPlace(direction, index);
    //Place of the desired cell in local arrays
```



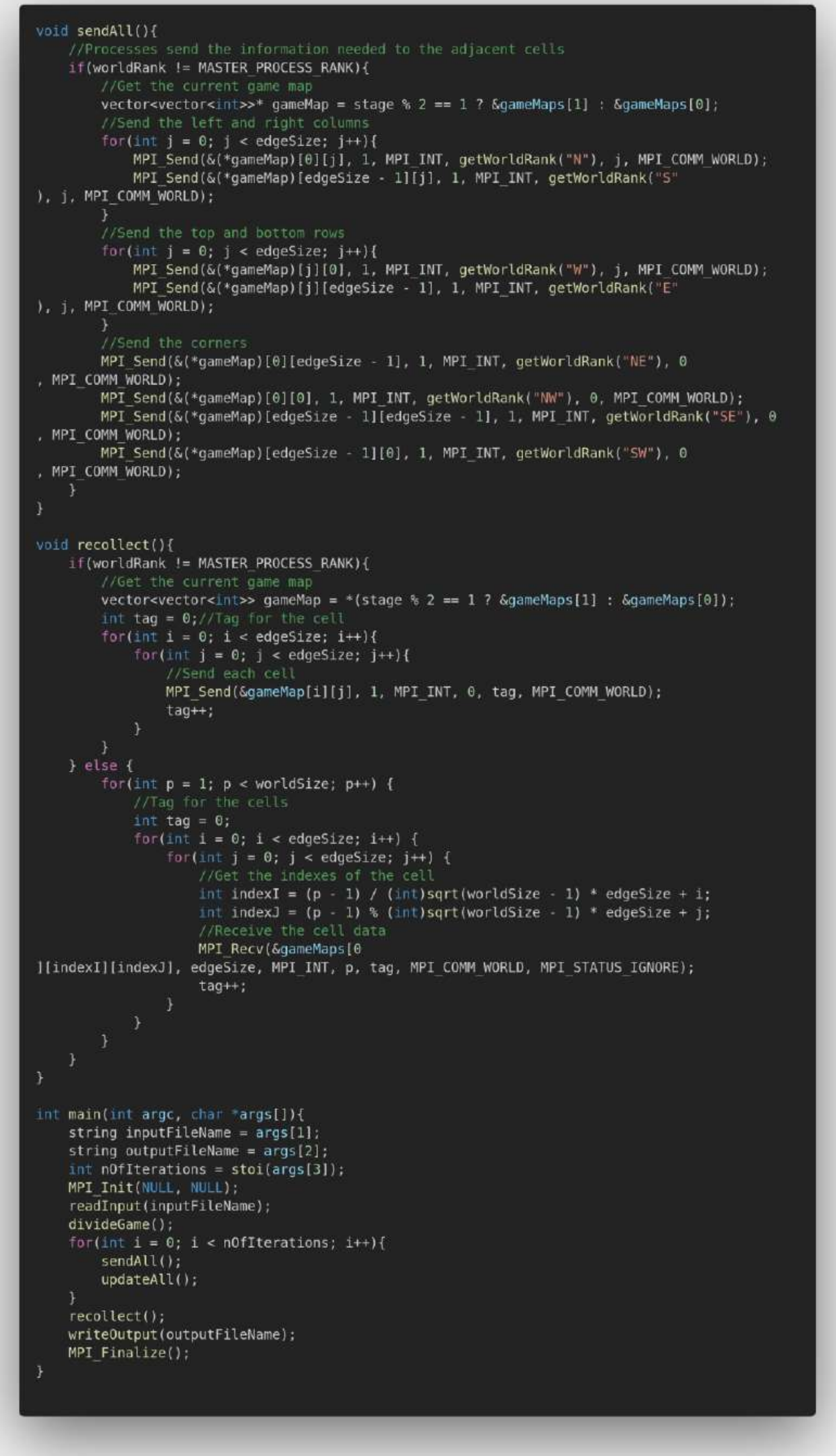
```
    if(direction.length() == 2){
        index = 0;
        //Tag will be zero if the direction is diagonal
    }
    if(*place == NOT_RECEIVED){//Neighbor cell is not received yet
        MPI_Recv(&result, 1, MPI_INT, getWorldRank
(direction), index, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //Receive the cell
        *place = result;
        //Store it in local array
    } else {
        result = *place;
        //Set the result from local array
    }
    return result;
}

int nOfLivingNeighbors(int i, int j){
    //Returns the number of living neighbors of the cell
    vector<vector<int>>* gameMap = stage % 2 == 1 ? &gameMaps[1] : &gameMaps[0];
    //Map to be used is chosen by looking at the stage
    int lastIndex = edgeSize - 1;//Last column index
    int N, NE, NW, W, E, S, SE, SW;//Neighbors in directions
    if(i == 0){//i is in the first row
        N = getFromArrayOrReceive("N", j);
        if(j == 0) {//j is in the first column
            NW = getFromArrayOrReceive("NW", j - 1);
        } else {
            NW = getFromArrayOrReceive("N", j - 1);
        }
        if(j == lastIndex) {//j is in the last column
            NE = getFromArrayOrReceive("NE", j + 1);
        } else {
            NE = getFromArrayOrReceive("N", j + 1);
        }
    } else {
        N = (*gameMap)[i - 1][j];
        if(j == 0){//j is in the first column
            NW = getFromArrayOrReceive("W", i - 1);
        } else {
            NW = (*gameMap)[i - 1][j - 1];
        }
        if(j == lastIndex){//j is in the last column
            NE = getFromArrayOrReceive("E", i - 1);
        } else {
            NE = (*gameMap)[i - 1][j + 1];
        }
    }
    if(i == lastIndex){//i is in the last row
        S = getFromArrayOrReceive("S", j);
        if(j == 0) {//j is in the first column
            SW = getFromArrayOrReceive("SW", j - 1);
        } else {
            SW = getFromArrayOrReceive("S", j - 1);
        }
        if(j == lastIndex) {//j is in the last column
            SE = getFromArrayOrReceive("SE", j + 1);
        } else {
            SE = getFromArrayOrReceive("S", j + 1);
        }
    } else {
        S = (*gameMap)[i + 1][j];
        if(j == 0){//j is in the first column
            SW = getFromArrayOrReceive("W", i + 1);
```



```cpp
        } else {
            SW = (*gameMap)[i + 1][j - 1];
        }
        if(j == lastIndex){//j is in the last column
            SE = getFromArrayOrReceive("E", i + 1);
        } else {
            SE = (*gameMap)[i + 1][j + 1];
        }
    }
    if(j == lastIndex){//j is in the last column
        E = getFromArrayOrReceive("E", i);
    } else {
        E = (*gameMap)[i][j + 1];
    }
    if(j == 0){//j is in the first column
        W = getFromArrayOrReceive("W", i);
    } else {
        W = (*gameMap)[i][j - 1];
    }
    return N + NE + NW + S + SE + SW + E + W;
}

void update(int i, int j){
    int livingNeighboorCount = nOfLivingNeighbors(i, j);
    //Select the old game map and the new game map by looking at the stage
    vector<vector<int>>* newGameMap = stage % 2 == 1 ? &gameMaps[0] : &gameMaps[1];
    vector<vector<int>>* oldGameMap = stage % 2 == 1 ? &gameMaps[1] : &gameMaps[0];
    if(livingNeighboorCount < 2 || livingNeighboorCount > 3){
        (*newGameMap)[i][j] = 0;
        //Overpopulation or loneliness
    } else if(livingNeighboorCount == 3){
        (*newGameMap)[i][j] = 1;
        //Reproduce
    } else {
        (*newGameMap)[i][j] = (*oldGameMap)[i][j];
    }
}

void readInput(string filename){
    //Read input to the game map array
    if(worldRank == MASTER_PROCESS_RANK){
        gameMaps.push_back(vector<vector<int>>());
        string line;
        ifstream inputFile;
        inputFile.open(filename);
        while (getline(inputFile, line)){
            gameMaps[0].push_back(vector<int>());
            for(int i = 0; i < line.length() - 1; i+=2){
                gameMaps[0][gameMaps[0].size() - 1].push_back(line[i] - '0');
                //Store chars as int in gameMap array
            }
        }
        inputFile.close();
    }
}

void writeOutput(string filename){
    //Write the game map to the output file
    if(worldRank == MASTER_PROCESS_RANK){
        ofstream outputFile;
        outputFile.open(filename);
        for(int i = 0; i < gameMaps[0].size(); i++){
            for(int j = 0; j < gameMaps[0][i].size(); j++){
                outputFile << gameMaps[0][i][j] << " ";
            }
            outputFile << endl;
        }
        outputFile.close();
    }
}
```



```
void updateAll() {
    if(worldRank != MASTER_PROCESS_RANK){
        for(int i = 0; i < edgeSize; i++){
            for(int j = 0; j < edgeSize; j++){
                update(i, j);//Update all cells of game map
            }
        }
        //Reinitialize neighbor vectors
        ups = vector<int>(edgeSize, NOT_RECEIVED);
        downs = vector<int>(edgeSize, NOT_RECEIVED);
        lefts = vector<int>(edgeSize, NOT_RECEIVED);
        rights = vector<int>(edgeSize, NOT_RECEIVED);
        corners = vector<int>(4, NOT_RECEIVED);
        stage++;//Increment the stage
    }
}

void divideGame(){
    //Divide the game to the processes
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);//get world size
    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);//get world rank
    if(worldRank == MASTER_PROCESS_RANK){
        edgeSize = gameMaps[0].size() / (int)sqrt(worldSize - 1);//set edge size
        for(int i = 1; i < worldSize; i++){
            int tag = 1;//Tag for cells
            int j = (i - 1) / (int)sqrt(worldSize - 1) * edgeSize;
            //Beginning of the row index
            MPI_Send(&edgeSize, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            //Send edge size to the process
            int boundaryJ = j + edgeSize;
            //Boundary of the row
            for(; j < boundaryJ; j++){
                int k = (i - 1) % (int)sqrt(worldSize - 1) * edgeSize;
                //Beginning of the column index
                int boundaryK = k + edgeSize;
                //Boundary of the column
                for(; k < boundaryK; k++){
                    //Send the cell and increment the tag
                    MPI_Send(&gameMaps[0][j][k], 1, MPI_INT, i, tag, MPI_COMM_WORLD);
                    tag++;
                }
            }
        }
    } else {
        //Initialize the maps
        gameMaps = vector<vector<vector<int>>>(2);
        //Receive the edge size
        MPI_Recv(&edgeSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //Set the tag for the cells
        int tag = 1;
        //Initalize the neighbor vectors
        ups = vector<int>(edgeSize, NOT_RECEIVED);
        downs = vector<int>(edgeSize, NOT_RECEIVED);
        lefts = vector<int>(edgeSize, NOT_RECEIVED);
        rights = vector<int>(edgeSize, NOT_RECEIVED);
        corners = vector<int>(4, NOT_RECEIVED);
        for(int i = 0; i < edgeSize; i++){
            //Push empty vectors to fill as rows
            gameMaps[0].push_back(vector<int>());
            gameMaps[1].push_back(vector<int>());
            for(int j=0; j < edgeSize; j++){
                //get new value of the cell
                int newVal;
                MPI_Recv(&newVal, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                //Set the value as the cell of the arrays
                gameMaps[0][i].push_back(newVal);
                gameMaps[1][i].push_back(newVal);
                //Increment the tag for next cell
                tag++;
            }
        }
    }
}
```



```
void sendAll(){
    //Processes send the information needed to the adjacent cells
    if(worldRank != MASTER_PROCESS_RANK){
        //Get the current game map
        vector<vector<int>>* gameMap = stage % 2 == 1 ? &gameMaps[1] : &gameMaps[0];
        //Send the left and right columns
        for(int j = 0; j < edgeSize; j++){
            MPI_Send(&(*gameMap)[0][j], 1, MPI_INT, getWorldRank("N"), j, MPI_COMM_WORLD);
            MPI_Send(&(*gameMap)[edgeSize - 1][j], 1, MPI_INT, getWorldRank("S"
), j, MPI_COMM_WORLD);
        }
        //Send the top and bottom rows
        for(int j = 0; j < edgeSize; j++){
            MPI_Send(&(*gameMap)[j][0], 1, MPI_INT, getWorldRank("W"), j, MPI_COMM_WORLD);
            MPI_Send(&(*gameMap)[j][edgeSize - 1], 1, MPI_INT, getWorldRank("E"
), j, MPI_COMM_WORLD);
        }
        //Send the corners
        MPI_Send(&(*gameMap)[0][edgeSize - 1], 1, MPI_INT, getWorldRank("NE"), 0
, MPI_COMM_WORLD);
        MPI_Send(&(*gameMap)[0][0], 1, MPI_INT, getWorldRank("NW"), 0, MPI_COMM_WORLD);
        MPI_Send(&(*gameMap)[edgeSize - 1][edgeSize - 1], 1, MPI_INT, getWorldRank("SE"), 0
, MPI_COMM_WORLD);
        MPI_Send(&(*gameMap)[edgeSize - 1][0], 1, MPI_INT, getWorldRank("SW"), 0
, MPI_COMM_WORLD);
    }
}

void recollect(){
    if(worldRank != MASTER_PROCESS_RANK){
        //Get the current game map
        vector<vector<int>> gameMap = *(stage % 2 == 1 ? &gameMaps[1] : &gameMaps[0]);
        int tag = 0;//Tag for the cell
        for(int i = 0; i < edgeSize; i++){
            for(int j = 0; j < edgeSize; j++){
                //Send each cell
                MPI_Send(&gameMap[i][j], 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
                tag++;
            }
        }
    } else {
        for(int p = 1; p < worldSize; p++) {
            //Tag for the cells
            int tag = 0;
            for(int i = 0; i < edgeSize; i++) {
                for(int j = 0; j < edgeSize; j++) {
                    //Get the indexes of the cell
                    int indexI = (p - 1) / (int)sqrt(worldSize - 1) * edgeSize + i;
                    int indexJ = (p - 1) % (int)sqrt(worldSize - 1) * edgeSize + j;
                    //Receive the cell data
                    MPI_Recv(&gameMaps[0
][indexI][indexJ], edgeSize, MPI_INT, p, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    tag++;
                }
            }
        }
    }
}

int main(int argc, char *args[]){
    string inputFileName = args[1];
    string outputFileName = args[2];
    int nOfIterations = stoi(args[3]);
    MPI_Init(NULL, NULL);
    readInput(inputFileName);
    divideGame();
    for(int i = 0; i < nOfIterations; i++){
        sendAll();
        updateAll();
    }
    recollect();
    writeOutput(outputFileName);
    MPI_Finalize();
}
```