THE GAME OF LIFE

CMPE 300 Programming Project Report

Sertay AKPINAR
2016400075

December 22, 2019

INTRODUCTION:

The goal of my project is to simulate the Conway's Game of Life using a parallel algorithm via Message Passing Interface (MPI). The Game of Life is meant to show what happens to the neighbor organisms in each generation(iteration). The 8 cells that are immediately around a cell are considered as its neighbors. 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).

I used an implementation of Message Passing Interface (MPI) in C++. It provides to me with multiple processes to work with, and functions to send/receive messages among them for the data exchange.

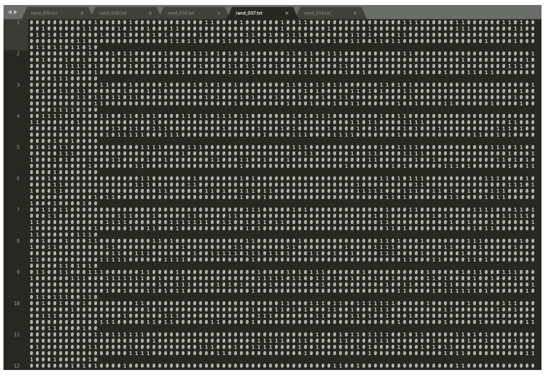
PROGRAM INTERFACE:

The project contains one C++ file called "game.cpp". It can be compiled into the executable game with the following*: "mpic++ game.cpp -o game".* The compiled MPI program game, can be run with the following: *"mpirun -np [M] –oversubscribe ./game input.txt output.txt [T]".* The program terminates itself. [M] is the number of processes to be executed and [T] is number of iterations. The program termination is the program's job, there nothing to do for user.

PROGRAM EXECUTION:

The inputs and outputs are a text written with the numbers of 0 and 1. User gives the command on the command line interpreter then the program gives the output after T iteration updating the input file using M processes.

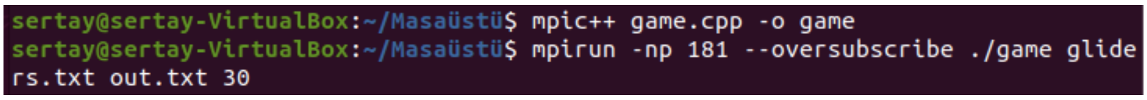
Output Example:



INPUT AND OUTPUT:

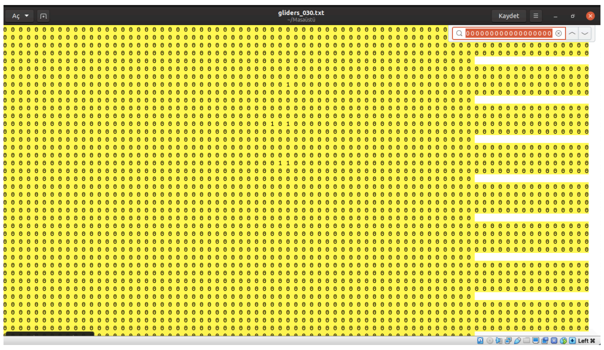
The following picture is a part of the input called "gliders.txt". Since it's too long, I added only a part of it.

The input syntax is “mpirun -np [M] --oversubscribe ./game input.txt output.txt [T]”:
[M] is the number of processes to run game on. The flag --oversubscribe allows you to set [M] more than just the number of logical cores on your machine. 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,



```
sertay@sertay-VirtualBox:~/Masaüstü$ mpic++ game.cpp -o game
sertay@sertay-VirtualBox:~/Masaüstü$ mpirun -np 181 --oversubscribe ./game glide
rs.txt out.txt 30
```

The following pictures are the desired outputs, if the user enters the command in the picture above. We can check the output’s correctness by copying the out.txt and searching the copied text in the desired outputs as I did in the example pictures. All the input and output files are in .txt format.

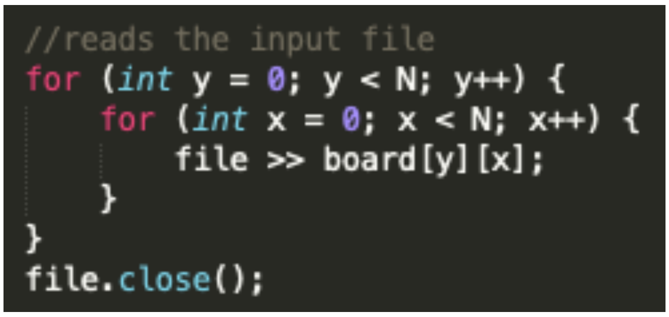


PROGRAM STRUCTURE:
This is an implementation of a MPI program in C++, so that I included “mpi.h” library in the code. I also included “iostream” and “fstream” for getting input and output.

- First, I initialized the variables that I will use during the implemantation.

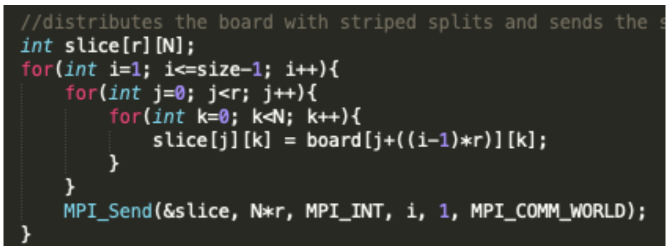
The main function has an if-else block. In the if block, I coded the duties of process 0. In the else block, I coded the duties of the other processes called “worker processes”.

IF BLOCK:



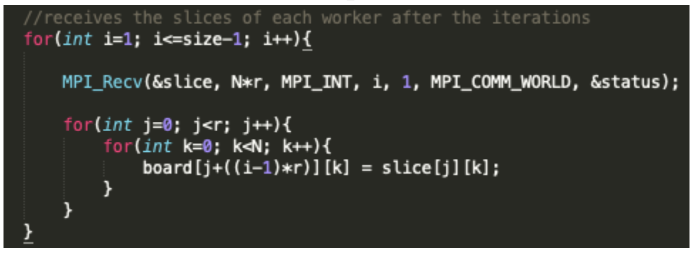
```
//reads the input file
for (int y = 0; y < N; y++) {
    for (int x = 0; x < N; x++) {
        file >> board[y][x];
    }
}
file.close();
```

- I read the input matrix with 2 for loops, after that I closed the file.



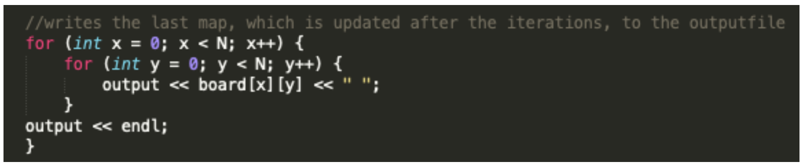
```
//distributes the board with striped splits and sends the
int slice[r][N];
for(int i=1; i<=size-1; i++){
    for(int j=0; j<r; j++){
        for(int k=0; k<N; k++){
            slice[j][k] = board[j+((i-1)*r)][k];
        }
    }
    MPI_Send(&slice, N*r, MPI_INT, i, 1, MPI_COMM_WORLD);
}
```

- I divided the map into the slices. I use “MPI_Send” function to distribute these slice matrices to the worker process.



```
//receives the slices of each worker after the iterations
for(int i=1; i<=size-1; i++){

    MPI_Recv(&slice, N*r, MPI_INT, i, 1, MPI_COMM_WORLD, &status);

    for(int j=0; j<r; j++){
        for(int k=0; k<N; k++){
            board[j+((i-1)*r)][k] = slice[j][k];
        }
    }
}
```

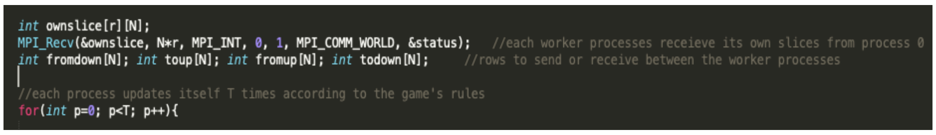
- Process 0 has to receive the slices after iterations. At this point, process 0 waits for the send signals from workers. After it receives the slices of each worker, it reunites the slices to obtain the updated game map. There is a basic math calculations for finding the corresponding index for the each cell of the slice in the map. $\left(\left[j + \left((i-1) * r\right)\right]\right)$



```
//writes the last map, which is updated after the iterations, to the outputfile
for (int x = 0; x < N; x++) {
    for (int y = 0; y < N; y++) {
        output << board[x][y] << " ";
    }
output << endl;
}
```

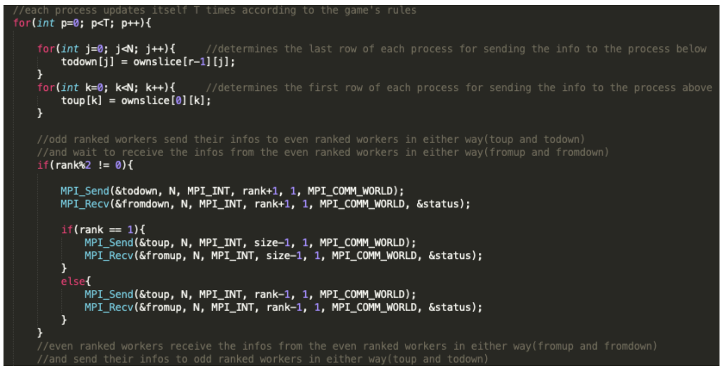
- I get the output matrix with 2 for loops. At the end of the code I closed the output.

<u>ELSE BLOCK:</u>



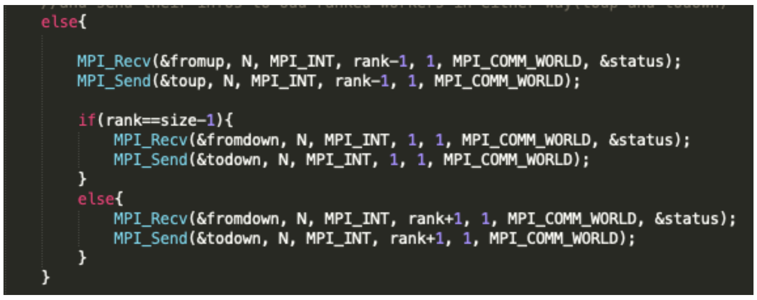
```
int ownslice[r][N];
MPI_Recv(&ownslice, N*r, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);   //each worker processes receieve its own slices from process 0
int fromdown[N]; int toup[N]; int fromup[N]; int todown[N];     //rows to send or receive between the worker processes

//each process updates itself T times according to the game's rules
for(int p=0; p<T; p++){
```

- “ownslice” is the each slice that a worker has.
- The workers receive their own slices from process 0.
- Direction arrays are the rows to send or to receive. (Ex: Row to receive from lower process is fromdown[N]. Row to send to upper process is toup[N])



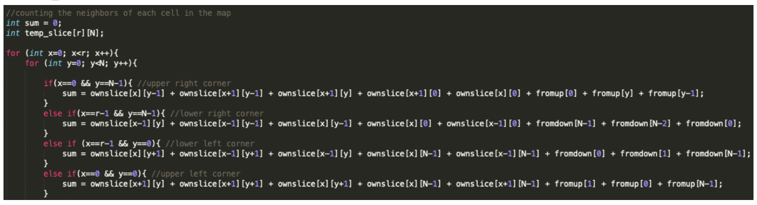
```
//each process updates itself T times according to the game's rules
for(int p=0; p<T; p++){

    for(int j=0; j<N; j++){     //determines the last row of each process for sending the info to the process below
        todown[j] = ownslice[r-1][j];
    }
    for(int k=0; k<N; k++){     //determines the first row of each process for sending the info to the process above
        toup[k] = ownslice[0][k];
    }

    //odd ranked workers send their infos to even ranked workers in either way(toup and todown)
    //and wait to receive the infos from the even ranked workers in either way(fromup and fromdown)
    if(rank%2 != 0){

        MPI_Send(&todown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD);
        MPI_Recv(&fromdown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD, &status);

        if(rank == 1){
            MPI_Send(&toup, N, MPI_INT, size-1, 1, MPI_COMM_WORLD);
            MPI_Recv(&fromup, N, MPI_INT, size-1, 1, MPI_COMM_WORLD, &status);
        }
        else{
            MPI_Send(&toup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD);
            MPI_Recv(&fromup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD, &status);
        }
    }
    //even ranked workers receive the infos from the even ranked workers in either way(fromup and fromdown)
    //and send their infos to odd ranked workers in either way(toup and todown)
```

- The main game mechanism (communication between the workers, calculating the neighbors etc.) is progressing in the scope of first for loop. This loop continues till the iterations ends. In other other words, when each slices reached its final status.
- The odd processes first send their info to the even processes then waits for the info from even processes.



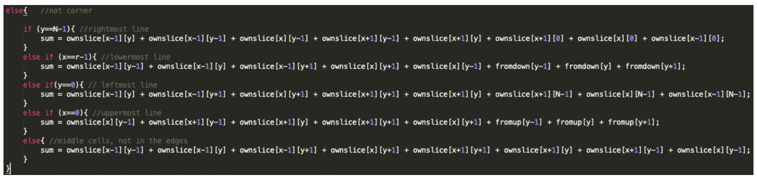
```
else{

    MPI_Recv(&fromup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD, &status);
    MPI_Send(&toup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD);

    if(rank==size-1){
        MPI_Recv(&fromdown, N, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
        MPI_Send(&todown, N, MPI_INT, 1, 1, MPI_COMM_WORLD);
    }
    else{
        MPI_Recv(&fromdown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD, &status);
        MPI_Send(&todown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD);
    }
}
```

- The even processes, first receive the info from the odd processes then send their info to odd processes.

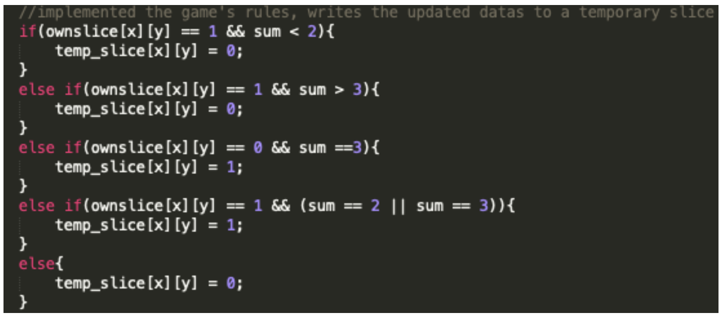


```
//counting the neighbors of each cell in the map
int sum = 0;
int temp_slice[r][N];

for (int x=0; x<r; x++){
    for (int y=0; y<N; y++){

        if(x==0 && y==N-1){ //upper right corner
            sum = ownslice[x][y-1] + ownslice[x+1][y-1] + ownslice[x+1][y] + ownslice[x+1][0] + ownslice[x][0] + fromup[0] + fromup[y] + fromup[y-1];
        }
        else if(x==r-1 && y==N-1){ //lower right corner
            sum = ownslice[x-1][y] + ownslice[x-1][y-1] + ownslice[x][y-1] + ownslice[x][0] + ownslice[x-1][0] + fromdown[N-1] + fromdown[N-2] + fromdown[0];
        }
        else if (x==r-1 && y==0){ //lower left corner
            sum = ownslice[x][y+1] + ownslice[x-1][y+1] + ownslice[x-1][y] + ownslice[x][N-1] + ownslice[x-1][N-1] + fromdown[0] + fromdown[1] + fromdown[N-1];
        }
        else if(x==0 && y==0){ //upper left corner
            sum = ownslice[x+1][y] + ownslice[x+1][y+1] + ownslice[x][y+1] + ownslice[x][N-1] + ownslice[x+1][N-1] + fromup[1] + fromup[0] + fromup[N-1];
        }
```

- Sum is for calculating the neighbors.
- Temp slice is for copying data and applying the changes in the data.
- I calculated all the cells with this if-else blocks in the pictures below and above. Used fromup and fromdown arrays in the necessary parts. (if I need a data of the other process.)

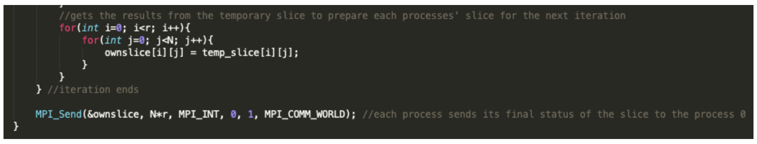


```
else{   //not corner

    if (y==N-1){ //rightmost line
        sum = ownslice[x-1][y] + ownslice[x-1][y-1] + ownslice[x][y-1] + ownslice[x+1][y-1] + ownslice[x+1][y] + ownslice[x+1][0] + ownslice[x][0] + ownslice[x-1][0];
    }
    else if (x==r-1){ //lowermost line
        sum = ownslice[x-1][y-1] + ownslice[x-1][y] + ownslice[x-1][y+1] + ownslice[x][y+1] + ownslice[x][y-1] + fromdown[y-1] + fromdown[y] + fromdown[y+1];
    }
    else if(y==0){ // leftmost line
        sum = ownslice[x-1][y] + ownslice[x-1][y+1] + ownslice[x][y+1] + ownslice[x+1][y+1] + ownslice[x+1][y] + ownslice[x+1][N-1] + ownslice[x][N-1] + ownslice[x-1][N-1];
    }
    else if (x==0){ //uppermost line
        sum = ownslice[x][y-1] + ownslice[x+1][y-1] + ownslice[x+1][y] + ownslice[x+1][y+1] + ownslice[x][y+1] + fromup[y-1] + fromup[y] + fromup[y+1];
    }
    else{ //middle cells, not in the edges
        sum = ownslice[x-1][y-1] + ownslice[x-1][y] + ownslice[x-1][y+1] + ownslice[x][y+1] + ownslice[x+1][y+1] + ownslice[x+1][y] + ownslice[x+1][y-1] + ownslice[x][y-1];
    }
}
```



```
//implemented the game's rules, writes the updated datas to a temporary slice
if(ownslice[x][y] == 1 && sum < 2){
    temp_slice[x][y] = 0;
}
else if(ownslice[x][y] == 1 && sum > 3){
    temp_slice[x][y] = 0;
}
else if(ownslice[x][y] == 0 && sum ==3){
    temp_slice[x][y] = 1;
}
else if(ownslice[x][y] == 1 && (sum == 2 || sum == 3)){
    temp_slice[x][y] = 1;
}
else{
    temp_slice[x][y] = 0;
}
```

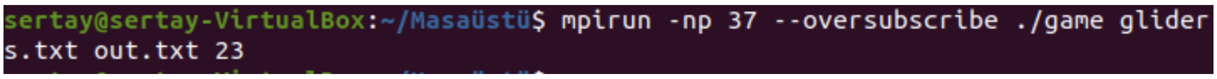
- Coded the game rules, in case of death, live and reproduce.



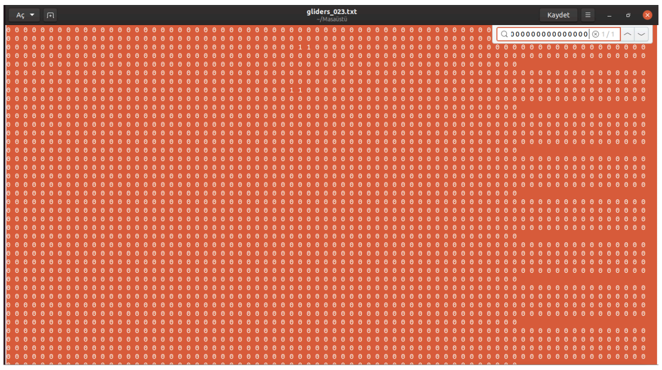
```
        //gets the results from the temporary slice to prepare each processes' slice for the next iteration
        for(int i=0; i<r; i++){
            for(int j=0; j<N; j++){
                ownslice[i][j] = temp_slice[i][j];
            }
        }
    } //iteration ends

    MPI_Send(&ownslice, N*r, MPI_INT, 0, 1, MPI_COMM_WORLD); //each process sends its final status of the slice to the process 0
}
```

- Updates each processes' slice according the data in the temp matrix.
- After the iteration ends, workers can send their final status of the slices to process 0.
  - At the end, I finalize the MPI and close the output.

Ex: 37 processes with 23 iterations, (36 workers) – It finds the match.



```
sertay@sertay-VirtualBox:~/Masaüstü$ mpirun -np 37 --oversubscribe ./game glider
s.txt out.txt 23
```



IMPROVEMENTS AND EXTENSIONS:

Even I get the correct outputs, the code could be more modular. But still the total program's code length is nothing much but 188 lines including the spaces.

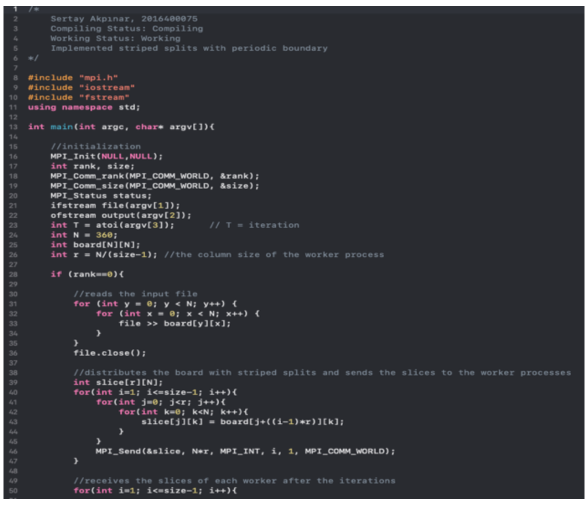
DIFFICULTIES ENCOUNTERED:

Since it's my first MPI programming project, it takes a bit time to comprehend the mechanism of MPI and the usage of MPI functions such as send and receive. The other point is, implementing it boundary. This also takes a bit time but nothing more the comprehension of the MPI. I only add some row arrays to get data from the other processes such as fromdown, toup etc.

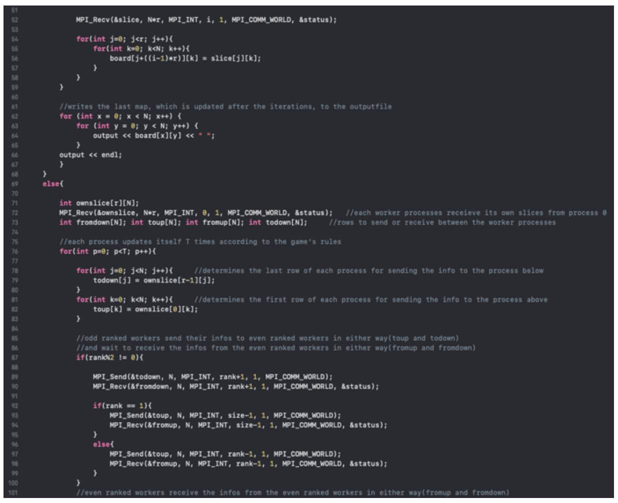
CONCLUSION:

As seen from the outputs, the program is able to handle the requested task correctly. I don't get any compiler errors and I get the requested outputs. I implemented the code in striped splits way with periodic boundary.

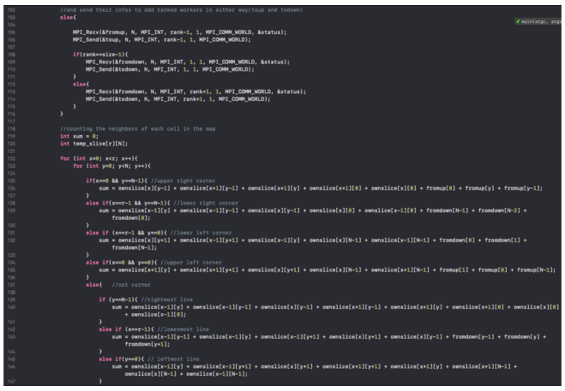
APPENDICES:



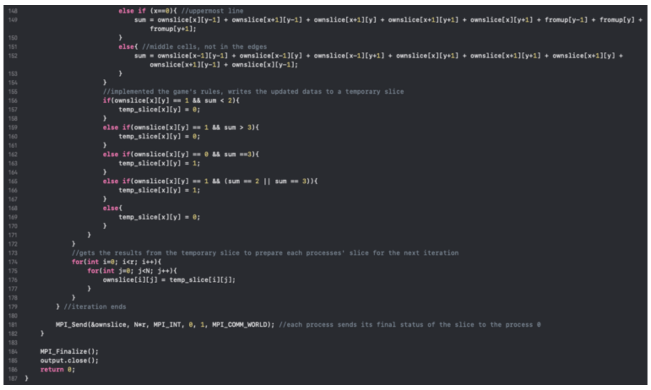
```cpp
/*
    Sertay Akpınar, 2016400075
    Compiling Status: Compiling
    Working Status: Working
    Implemented striped splits with periodic boundary
*/

#include "mpi.h"
#include "iostream"
#include "fstream"
using namespace std;

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

    //initialization
    MPI_Init(NULL,NULL);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Status status;
    ifstream file(argv[1]);
    ofstream output(argv[2]);
    int T = atoi(argv[3]);      // T = iteration
    int N = 360;
    int board[N][N];
    int r = N/(size-1); //the column size of the worker process

    if (rank==0){

        //reads the input file
        for (int y = 0; y < N; y++) {
            for (int x = 0; x < N; x++) {
                file >> board[y][x];
            }
        }
        file.close();

        //distributes the board with striped splits and sends the slices to the worker processes
        int slice[r][N];
        for(int i=1; i<=size-1; i++){
            for(int j=0; j<r; j++){
                for(int k=0; k<N; k++){
                    slice[j][k] = board[j+((i-1)*r)][k];
                }
            }
            MPI_Send(&slice, N*r, MPI_INT, i, 1, MPI_COMM_WORLD);
        }

        //receives the slices of each worker after the iterations
        for(int i=1; i<=size-1; i++){
```



```cpp

            MPI_Recv(&slice, N*r, MPI_INT, i, 1, MPI_COMM_WORLD, &status);

            for(int j=0; j<r; j++){
                for(int k=0; k<N; k++){
                    board[j+((i-1)*r)][k] = slice[j][k];
                }
            }
        }

        //writes the last map, which is updated after the iterations, to the outputfile
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                output << board[x][y] << " ";
            }
        output << endl;
        }
    }
    else{

        int ownslice[r][N];
        MPI_Recv(&ownslice, N*r, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);   //each worker processes receieve its own slices from process 0
        int fromdown[N]; int toup[N]; int fromup[N]; int todown[N];     //rows to send or receive between the worker processes

        //each process updates itself T times according to the game's rules
        for(int p=0; p<T; p++){

            for(int j=0; j<N; j++){     //determines the last row of each process for sending the info to the process below
                todown[j] = ownslice[r-1][j];
            }
            for(int k=0; k<N; k++){     //determines the first row of each process for sending the info to the process above
                toup[k] = ownslice[0][k];
            }

            //odd ranked workers send their infos to even ranked workers in either way(toup and todown)
            //and wait to receive the infos from the even ranked workers in either way(fromup and fromdown)
            if(rank%2 != 0){

                MPI_Send(&todown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD);
                MPI_Recv(&fromdown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD, &status);

                if(rank == 1){
                    MPI_Send(&toup, N, MPI_INT, size-1, 1, MPI_COMM_WORLD);
                    MPI_Recv(&fromup, N, MPI_INT, size-1, 1, MPI_COMM_WORLD, &status);
                }
                else{
                    MPI_Send(&toup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD);
                    MPI_Recv(&fromup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD, &status);
                }
            }
            //even ranked workers receive the infos from the even ranked workers in either way(fromup and fromdown)
```



```
            //and send their infos to odd ranked workers in either way(toup and todown)
            else{

                MPI_Recv(&fromup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD, &status);
                MPI_Send(&toup, N, MPI_INT, rank-1, 1, MPI_COMM_WORLD);

                if(rank==size-1){
                    MPI_Recv(&fromdown, N, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
                    MPI_Send(&todown, N, MPI_INT, 1, 1, MPI_COMM_WORLD);
                }
                else{
                    MPI_Recv(&fromdown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD, &status);
                    MPI_Send(&todown, N, MPI_INT, rank+1, 1, MPI_COMM_WORLD);
                }
            }

            //counting the neighbors of each cell in the map
            int sum = 0;
            int temp_slice[r][N];

            for (int x=0; x<r; x++){
                for (int y=0; y<N; y++){

                    if(x==0 && y==N-1){ //upper right corner
                        sum = ownslice[x][y-1] + ownslice[x+1][y-1] + ownslice[x+1][y] + ownslice[x+1][0] + ownslice[x][0] + fromup[0] + fromup[y] + fromup[y-1];
                    }
                    else if(x==r-1 && y==N-1){ //lower right corner
                        sum = ownslice[x-1][y] + ownslice[x-1][y-1] + ownslice[x][y-1] + ownslice[x][0] + ownslice[x-1][0] + fromdown[N-1] + fromdown[N-2] +
                            fromdown[0];
                    }
                    else if (x==r-1 && y==0){ //lower left corner
                        sum = ownslice[x][y+1] + ownslice[x-1][y+1] + ownslice[x-1][y] + ownslice[x][N-1] + ownslice[x-1][N-1] + fromdown[0] + fromdown[1] +
                            fromdown[N-1];
                    }
                    else if(x==0 && y==0){ //upper left corner
                        sum = ownslice[x+1][y] + ownslice[x+1][y+1] + ownslice[x][y+1] + ownslice[x][N-1] + ownslice[x+1][N-1] + fromup[1] + fromup[0] + fromup[N-1];
                    }
                    else{   //not corner

                        if (y==N-1){ //rightmost line
                            sum = ownslice[x-1][y] + ownslice[x-1][y-1] + ownslice[x][y-1] + ownslice[x+1][y-1] + ownslice[x+1][y] + ownslice[x+1][0] + ownslice[x][0]
                                + ownslice[x-1][0];
                        }
                        else if (x==r-1){ //lowermost line
                            sum = ownslice[x-1][y-1] + ownslice[x-1][y] + ownslice[x-1][y+1] + ownslice[x][y+1] + ownslice[x][y-1] + fromdown[y-1] + fromdown[y] +
                                fromdown[y+1];
                        }
                        else if(y==0){ // leftmost line
                            sum = ownslice[x-1][y] + ownslice[x-1][y+1] + ownslice[x][y+1] + ownslice[x+1][y+1] + ownslice[x+1][y] + ownslice[x+1][N-1] +
                                ownslice[x][N-1] + ownslice[x-1][N-1];
                        }
```



```
                        else if (x==0){ //uppermost line
                            sum = ownslice[x][y-1] + ownslice[x+1][y-1] + ownslice[x+1][y] + ownslice[x+1][y+1] + ownslice[x][y+1] + fromup[y-1] + fromup[y] +
                                fromup[y+1];
                        }
                        else{ //middle cells, not in the edges
                            sum = ownslice[x-1][y-1] + ownslice[x-1][y] + ownslice[x-1][y+1] + ownslice[x][y+1] + ownslice[x+1][y+1] + ownslice[x+1][y] +
                                ownslice[x+1][y-1] + ownslice[x][y-1];
                        }
                    }
                    //implemented the game's rules, writes the updated datas to a temporary slice
                    if(ownslice[x][y] == 1 && sum < 2){
                        temp_slice[x][y] = 0;
                    }
                    else if(ownslice[x][y] == 1 && sum > 3){
                        temp_slice[x][y] = 0;
                    }
                    else if(ownslice[x][y] == 0 && sum ==3){
                        temp_slice[x][y] = 1;
                    }
                    else if(ownslice[x][y] == 1 && (sum == 2 || sum == 3)){
                        temp_slice[x][y] = 1;
                    }
                    else{
                        temp_slice[x][y] = 0;
                    }
                }
            }
            //gets the results from the temporary slice to prepare each processes' slice for the next iteration
            for(int i=0; i<r; i++){
                for(int j=0; j<N; j++){
                    ownslice[i][j] = temp_slice[i][j];
                }
            }
        } //iteration ends

        MPI_Send(&ownslice, N*r, MPI_INT, 0, 1, MPI_COMM_WORLD); //each process sends its final status of the slice to the process 0
    }

    MPI_Finalize();
    output.close();
    return 0;
}
```