MPI Project: Parallel Game of Life

Course ID and Name: CmpE 300 – Analysis of Algorithms

Name of the Student: Halil Umut ÖZDEMİR

Student ID of the Student: 2016400168

Type of the Project: Programming Project

Submission Date: 20.12.2019

1. Introduction

In this project, the problem is to implement a Cellular Automata system with using Message Passing Interface (MPI). Our focus is on the cellular automation called the (Conway's) Game of Life, devised by J. H. Conway in 1970. In this game, there is a 2-dimensional orthogonal grid as a map. Each cell on the map either contain a creature or be empty.

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

- Loneliness Kill: A creature dies if it has less than 2 neighboring cells.
- Overpopulation Kill: A creature dies if it has more than 3 neighboring creatures.
- Reproduction: A new life appears on an empty cell if it has exactly 3 neighboring creatures.
- For all other conditions, the creatures remain alive, and the empty spaces remain empty.

Because this is a cellular automata system. There are 3 rules about cellular automata:

- The map at time t = 0 is as initialized.
- To calculate the map's state at time t = 1, we look at the state at time t = 0. Foreach cell on the map at time t = 1, we look at the same cell and its neighbors at time t = 0.
- We keep on simulating like this until time t = T.

2. Program Interface

As it is written in Introduction section, the simulation program was coded using a parallel algorithm. To maintain the parallelism, Message Passing Interface was used. So, to run the program Message Passing Interface (MPI) standard called OpenMPI must be installed in your runtime environment.

The source code of the program can be compiled with the following command.

mpic++ game.cpp -o game

The compiled MPI program game, can be run using the following command.

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

In this command [M] is the number of processes which runs during the program execution. Because I used the Checkered approach for splits number of processes must be $c^2 + 1$ where c is an even number. [T] is number of time steps of the

simulation. Other two parameters input.txt and output.txt are input and output of the program respectively.

If you want to terminate the program before its normal termination. You can use Ctrl+C to terminate the program. But if program is terminated with using this, the output file will not be generated.

3. Program Execution

To run the program, a user must give an input file name, an output file name and number of time steps to the program as an argument. Also, the number of processes which will be active during the execution of the program must be given to the mpirun command as an argument.

The number of time steps must be a non-negative integer. If it is a negative integer program takes it as 0. This parameter must be written using the decimal digits. For example, to execute the simulation for ten time steps [T] parameter at the command in the Program Interface Section must be 10 not ‘ten’ or ‘Ten’ etc.

The input file must be in the same directory with the executable file. Also, only the name of the input file will be given to the program as an argument. The format of the input file will be explained in the Input and Output Section (next section).

At the end of the execution, program will create an output file with the name that is given as an argument to the program. The output file name parameter is only the name of the output file. Program will create the output file if such a file does not exist in the directory of the executable. If it exists program will overwrite the file that is given as an argument. So, please be careful when writing the output file name. You may lose data if wrong file name was given as an argument. The format of the output file will be explained in the Input and Output Section (next section).

The last argument of the program is number of processes which will be active during the execution of the program. Because I used the Checkered approach for splits number of processes must be $c^2 + 1$ where c is an even number. Any other number which violates this rule may create errors and wrong outputs.

The functionality of the program is to simulate Game of Life, which is explained in the Introduction Section. Program reads the input file, which is the state of your map at the time 0. Then It simulates the Game of Life with the its rules, which are explained in the Introduction Section, for number of time steps that is given as an argument to the program. During the execution, the neighbors of the cells which haven’t got 8 adjacent cells are considered as in the Periodic approach. In this approach their neighbors are not missing. They are just the opposite end of the map. Because of this approach the map is called toroidal. At the end of the execution, program gives the output to a file which is one of the arguments of the program. The output file shows the state of the map at the time step you want. The formats of the input and output file will be explained in the Input and Output Section (next section).

4. Input and Output

In this section the format of the input and output files are explained. The input file shows the initial state of the map and the output file shows the last state of the map. There are some rules about the format of the input and output files:

- The map size must be 360x360.
- The rows must be separated by a single new line (\n) character.
- Each cell on a row must be separated by a single space character.
- Each cell must contain only 0 or 1. 1 indicates living creature and 0 indicates empty cell.

360x360 example is very large so the example input, and output is given as 6x6.

1 1 0 0 1 0	1 0 1 0 1 0
0 0 1 0 0 1	1 0 0 0 1 0
1 0 1 0 0 1	0 1 1 0 1 0
0 1 0 0 0 1	1 0 0 0 1 1
0 0 1 1 1 1	1 0 1 0 0 0
0 0 1 0 1 1	1 0 1 0 1 0
Example Input File	Example Output File (After 10 time steps)

5. Program Structure

This project is the parallel implementation of the Game of Life with using OpenMPI. Because of this there are several topics about program structure will be explained in following subsections which are job hierarchy, splitting of the map, communication mechanism between processes and computation part of the worker processes.

A. Job Hierarchy

In this project the processes have a manager worker-relation between them. The process with the 0 rank is the manager process. It is responsible from the File input and output. Any other process doesn't make any file input or output. Also, it splits the data into disjoint parts and sending these parts to worker processes. At the end of the execution of all the worker processes, manager process takes the parts of the results from worker processes and merge these parts to create whole output.

Any process which has rank different than 0 is a worker process. Worker processes are responsible from the parallel simulation of the Game of Life. During

these simulations they communicate with each other when it is necessary. At the end of the execution they send their part of the output to the manager process. The worker processes don't make any file input and output.

B. Splitting the Map

As it is explained in the previous subsection (Job Hierarchy), the map of the simulation is divided into disjoint parts There is a separate part for each worker process.

I use the Checkered approach to split the map. In the case of checkered approach, the map will be split into square areas. Each of these square areas has $S/\sqrt{C}$ rows and columns where S is the number of entries in the row and columns of the total map and C is the number of worker processes. Because of this approach, the number of processes must equal to $c^2 + 1$ where c is an even number which will divide 360.

Positions of the data of the workers for C=16 is as follows.

Rank 1	Rank 2	Rank 3	Rank 4
Rank 5	Rank 6	Rank 7	Rank 8
Rank 9	Rank 10	Rank 11	Rank 12
Rank 13	Rank 14	Rank 15	Rank 16

C. Communication Mechanism Between Processes

In the project design, there is 2 types of communications which are manager-worker and worker-worker communication. For all communications message passing is used and all message sends, and message receives are blocking send and receives.

Manager-worker communication has 2 parts which are at the beginning and end of the program execution respectively. In the first part, all worker processes make a blocking receive and manager process sends the separate map partitions of the worker processes one by one. This part is executed only once at the beginning of the execution and because it is a sequential implementation it makes exactly number of

workers time communication. In the second part of this communication, all worker processes make a blocking send of their part to the manager process. Manager process makes a blocking receive starting from the first worker process and takes the data from them sequentially. Also, in this part there is exactly the number of workers times communication.

In the case of worker-worker communication, there is a much complex communication scheme. In this type of the communication, a worker process takes the entries which are above, under, left, right and crosses of the part of the worker. Because I use the Periodic approach all worker processes must communicate with 8 of their neighbors for each time step. For example, the process with rank 1 communicates with process 2, process 5, process 6, process 13, process 14, process 16, process 4 and process 8. To obtain the most efficient and deadlock free communication, for every direction I divide the processes into 2 disjoint sets. For each direction one set sends the data and other set receives the data.

The worker-worker communication starts with right and left communication. For these 2 directions I separate the even and odd ranked processes into 2 disjoint sets. Firstly, odd processes send to their rightmost process and even processes are receiving from them. Then even processes send to the leftmost process and odd processes receives from them. This communication continues in the same way until all processes are completing their sends and receives.

Secondly, up and down communication is done. For this communication, the processes are divided as processes in odd rows and processes in even rows. The indexing of the rows starts from 0. Firstly, even row processes send data to the process below and odd row processes receives data from the process above. After that odd row processes sends data to processes below and even processes receives from the processes above. This communication continues in the same way until all processes are completing their sends and receives.

At the and cross communications occur. For this communication, the processes are divided as processes in odd rows and processes in even rows like previous communication. The indexing of the rows starts from 0. Firstly, even row processes send data to the bottom right process and odd row processes receives data from the upper left process. After that odd row processes sends data to the bottom right processes and even processes receives from the upper left processes. This communication continues in the same way until all processes are completing their sends and receives.

During all these communications, for processes which are at the edge of the map the rank of the process that the message is sent are calculated separately. To see these calculations, the comments of the code is enough. This is because the Periodic approach is used to implement the simulation.

For all these communications because always half of the processes send, and another half receives parallel. The number of sequential communications is 16 always which is constant which improves the efficiency of the communication. Also, all these communications must be done for all time steps. For detailed information about all

communications, comments in the code is enough to see all of the steps of the communications.

D. Computation Part of The Worker Processes

After the communication part of the worker processes, all worker process computes the next state of the part using the rules of Game of Life which are explained in the Introduction section. After the communication phase, all workers create a bigger 2-dimensional array (*full_input* array) and combines their separate part (*input* array) and the messages received from neighbors. At this step all messages are put to the correct place in the *full_input* array. For example, message received from the bottom process must be placed to the bottom of the *full_input* array. After that it calculates the total value of the neighbors of every entry of the separate part of the worker process. According to the rules of Game of Life explained in the Introduction Section.

For each time step all worker processes makes the communication phase and computation phase. At the end of the simulation of the last time step all worker processes sends the last state of their map to the manager process.

6. Examples

In this section I will give an example of a 6x6 grid for 4 time steps to see the execution of the program. For each turn the input of the turn is the last map state of the previous state.

Input-Turn 1:	Neighbor Values:	Last Map State:
1 1 0 0 1 0	3 3 3 5 3 5	1 1 1 0 1 0
0 0 1 0 0 1	5 6 3 4 3 5	0 0 1 0 1 0
1 0 1 0 0 1	3 4 2 3 4 3	1 0 1 1 0 1
0 1 0 0 0 1	5 3 5 5 6 4	0 1 0 0 0 0
0 0 1 1 1 1	3 4 3 4 3 4	1 0 1 0 1 0
0 0 1 0 1 1	4 6 3 6 3 5	0 0 1 0 1 0

Turn 2-Neighbor Values:	Last Map State:
1 3 3 6 2 2	0 1 1 0 1 0
3 6 4 5 3 5	1 0 0 0 1 0
3 4 2 2 3 4	1 0 1 1 1 0
3 5 3 6 3 6	1 0 1 0 1 0
1 4 2 6 2 4	0 0 1 0 1 0
1 4 3 6 2 3	0 0 1 0 1 1

Turn 3-Neighbor Values:	Last Map State:
3 4 1 4 3 5	1 0 0 0 1 0
2 4 2 5 3 6	1 0 0 0 1 0
2 5 2 5 2 6	1 0 1 0 1 0
1 4 2 6 2 4	0 0 1 0 1 0
1 3 3 6 3 4	0 1 1 0 1 0
3 5 2 5 3 5	1 0 1 0 1 0
Turn 4-Neighbor Values:	Output:
2 4 1 4 2 5	1 0 0 0 1 0
2 4 1 4 2 6	1 0 0 0 1 0
1 3 2 5 2 4	0 1 1 0 1 0
2 5 4 5 2 3	0 0 0 0 1 1
3 4 1 4 3 6	1 0 0 0 1 0
2 4 0 3 3 6	1 0 0 1 1 0

7. Improvements and Extensions

In this project, I think the weal part of my program is the communication mechanism between workers and manager. Because, as explained in Program Structure section, the communication between workers and manager is done sequentially. This situation increases the complexity of the algorithm of the program. This feature can be improved by adding a new layer between workers and manager. If a layer with $\sqrt{C}$ processes are added between manager and workers where C is the number of worker processes, the communication can be parallelized in this layer.

8. Difficulties Encountered

In this project the main difficulty is the deadlock problem of blocking send and blocking receive. To avoid the deadlocks, at any state of the program worker processes must be partitioned into 2 disjoint sets. One of the sets sends the data and another receives the data, because 2 sets are disjoint deadlock never occurs. Because the checkered approach is used to split the map, the rank calculations in these communications is also a difficulty in this project.

9. Conclusion

In conclusion, I learnt how to write a parallel program using OpenMPI and how the communication mechanism works in OpenMPI. In the case of algorithms, I learnt how to think as parallel and write parallel algorithms. So, I think this is a useful project to learn for me.

10. Appendices

```
//Halil Umut Özdemir,2016400168,Compiling,Working
#include <cstdio>
#include <iostream>
#include <fstream>
#include <mpi.h>
#include <math.h>

#define S 360
#define INPUT_MESSAGE 0
#define END_MESSAGE 1
#define FIRST 2 + 8*turn
#define SECOND 3 + 8*turn
#define THIRD 4 + 8*turn
#define FOURTH 5 + 8*turn
#define FIFTH 6 + 8*turn
#define SIXTH 7 + 8*turn
#define SEVENTH 8 + 8*turn
#define EIGHTH 9 + 8*turn

#define MASTER_PROCESS 0

using namespace std;

void combine_matrices(int length, void* input_par, void* full_input_par,int
* top, int* bottom, int* right, int* left, int top_right, int top_left, int
 bottom_right, int bottom_left){
    int (*input)[length] = static_cast<int (*)[length]>(input_par);
    int (*full_input)[length+2] = static_cast<int (*)[length+2]>(full_input
_par);
    for(int i=0;i<length+2;i++){
        for(int j=0;j<length+2;j++){
            if(i==0&&j==0)
                full_input[i][j] = top_left;
            else if(i==0&&j==length+1)
                full_input[i][j] = top_right;
            else if(i==length+1&&j==0)
                full_input[i][j] = bottom_left;
            else if(i==length+1&&j==length+1)
                full_input[i][j] = bottom_right;
            else if(i==0)
                full_input[i][j] = top[j-1];
            else if(i==length+1)
                full_input[i][j] = bottom[j-1];
            else if(j==0)
                full_input[i][j] = left[i-1];
            else if(j==length+1)
                full_input[i][j] = right[i-1];
```

```
            else
                full_input[i][j] = input[i-1][j-1];
        }
    }
}

void compute_next_state(int length, void* input_par, void* full_input_par){
    int (*input)[length] = static_cast<int (*)[length]>(input_par);
    int (*full_input)[length+2] = static_cast<int (*)[length+2]>(full_input
_par);
    for(int i=1;i<=length;i++){
        for(int j=1;j<=length;j++){
            int neigh = full_input[i-1][j-1] + full_input[i-
1][j] + full_input[i-1][j+1] + full_input[i][j-1] + full_input[i][j+1]
                        + full_input[i+1][j-
1] + full_input[i+1][j] + full_input[i+1][j+1];
                if(full_input[i][j]==1){
                    if(neigh<2){
                        input[i-1][j-1] = 0;
                    }else if(neigh>3){
                        input[i-1][j-1] = 0;
                    }
                }else{
                    if(neigh==3){
                        input[i-1][j-1] = 1;
                    }
                }
        }
    }
}

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

    int ierr = MPI_Init(&argc,&argv);
    int procid,numprocs;
    ierr = MPI_Comm_rank(MPI_COMM_WORLD,&procid);
    ierr = MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    int number_of_turns = stoi(argv[3]);
    numprocs--;
    int sqrt_numprocs = (int)sqrt(numprocs);
    int length = S/sqrt_numprocs;
    if(procid==MASTER_PROCESS){
        ifstream file;
        file.open(argv[1]);
        int input[sqrt_numprocs][length][length];
        for(int i=0;i<sqrt_numprocs;i++){
            for(int j=0;j<length;j++)
                for(int k=0;k<S;k++)
                    file >> input[k/length][j][k%length];
```

```
            for(int l=0;l<sqrt_numprocs;l++)
                MPI_Send(&input[l],length*length,MPI_INT,sqrt_numprocs*i+l+
1,INPUT_MESSAGE,MPI_COMM_WORLD);
        }
        int output[S][S];
        int temp[length][length];
        //Receives all data from worker processes
        for(int i=0;i<numprocs;i++){
            ierr = MPI_Recv(&temp,length*length,MPI_INT,i+1,END_MESSAGE,MPI
_COMM_WORLD,NULL);
            for(int j=0;j<length;j++)
                for(int k=0;k<length;k++)
                    output[(i/sqrt_numprocs)*length+j][(i%sqrt_numprocs)*le
ngth+k] = temp[j][k];
        }
        //Prints result to output file
        ofstream outfile;
        outfile.open(argv[2]);
        for(int i=0;i<S;i++){
            for(int j=0;j<S;j++)
                outfile << output[i][j] << " ";
            outfile << "\n";
        }
    }else{
        MPI_Status status;
        int input[length][length];
        ierr = MPI_Recv(&input,length*length,MPI_INT,MASTER_PROCESS,INPUT_M
ESSAGE,MPI_COMM_WORLD,&status);
        int top[length]; int bottom[length]; int right[length]; int left[le
ngth];
        int top_right; int bottom_right; int top_left; int bottom_left;
        for(int turn=0;turn < number_of_turns;turn++){
            //Right and Left Communication
            if(procid % 2){ //Odd processes
                int temp[length];
                //Copies its rightmost column
                for(int i=0;i<length;i++)
                    temp[i] = input[i][length-1];
                //Sends the rightmost column to the right process
                MPI_Send(&temp,length,MPI_INT,procid+1,FIRST,MPI_COMM_WORLD
);
                //Receives right from the right process
                ierr = MPI_Recv(&right,length,MPI_INT,procid+1,FIRST,MPI_CO
MM_WORLD,&status);
                //Reveives the left part from the left process
                if(procid%sqrt_numprocs==1)
                    ierr = MPI_Recv(&left,length,MPI_INT,procid+sqrt_numpro
cs-1,SECOND,MPI_COMM_WORLD,&status);
                else
```

```
                    ierr = MPI_Recv(&left,length,MPI_INT,procid-
1,SECOND,MPI_COMM_WORLD,&status);
                //Copies its leftmost column
                for(int i=0;i<length;i++)
                    temp[i] = input[i][0];
                //Sends the leftmost column to the left process
                if(procid%sqrt_numprocs==1)
                    MPI_Send(&temp,length,MPI_INT,procid+sqrt_numprocs-
1,SECOND,MPI_COMM_WORLD);
                else
                    MPI_Send(&temp,length,MPI_INT,procid-
1,SECOND,MPI_COMM_WORLD);

            }else{// Even Process
                //Reveives the left part from the left process
                ierr = MPI_Recv(&left,length,MPI_INT,procid-
1,FIRST,MPI_COMM_WORLD,&status);
                int temp[length];
                //Copies its leftmost column
                for(int i=0;i<length;i++)
                    temp[i] = input[i][0];
                //Sends the leftmost column to the left process
                MPI_Send(&temp,length,MPI_INT,procid-
1,FIRST,MPI_COMM_WORLD);
                //Copies its rightmost column
                for(int i=0;i<length;i++)
                    temp[i] = input[i][length-1];
                //Sends the rightmost column to the right process
                if(procid%sqrt_numprocs==0)
                    MPI_Send(&temp,length,MPI_INT,procid-
sqrt_numprocs+1,SECOND,MPI_COMM_WORLD);
                else
                    MPI_Send(&temp,length,MPI_INT,procid+1,SECOND,MPI_COMM_
WORLD);
                //Receives right from the right process
                if(procid%sqrt_numprocs==0)
                    ierr = MPI_Recv(&right,length,MPI_INT,procid-
sqrt_numprocs+1,SECOND,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&right,length,MPI_INT,procid+1,SECOND,M
PI_COMM_WORLD,&status);
            }
            //Up and Down Communication
            int row_index = (procid-1)/sqrt_numprocs;
            if(row_index % 2){//Odd Rows(Indexes starts from zero)
                //Receivs its top from top process
                ierr = MPI_Recv(&top,length,MPI_INT,procid-
sqrt_numprocs,THIRD,MPI_COMM_WORLD,&status);
                //Sends its down to down process
```

```
                if(row_index==sqrt_numprocs-1) //Last Row Process
                    MPI_Send(&input[length-
1],length,MPI_INT,procid+sqrt_numprocs-numprocs,THIRD,MPI_COMM_WORLD);
                else
                    MPI_Send(&input[length-
1],length,MPI_INT,procid+sqrt_numprocs,THIRD,MPI_COMM_WORLD);
                //Sends its up to up process
                MPI_Send(&input[0],length,MPI_INT,procid-
sqrt_numprocs,FOURTH,MPI_COMM_WORLD);
                //Reveives its bottom from bottom process
                if(row_index==sqrt_numprocs-1) //Last Row Process
                    ierr = MPI_Recv(&bottom,length,MPI_INT,procid+sqrt_nump
rocs-numprocs,FOURTH,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&bottom,length,MPI_INT,procid+sqrt_nump
rocs,FOURTH,MPI_COMM_WORLD,&status);
            }else{//Even Rows
                //Sends its down to the down process
                MPI_Send(&input[length-
1],length,MPI_INT,procid+sqrt_numprocs,THIRD,MPI_COMM_WORLD);
                //Receives its top from top process
                if(row_index==0)
                    ierr = MPI_Recv(&top,length,MPI_INT,procid-
sqrt_numprocs+numprocs,THIRD,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&top,length,MPI_INT,procid-
sqrt_numprocs,THIRD,MPI_COMM_WORLD,&status);
                //Receives its bottom from top process
                ierr = MPI_Recv(&bottom,length,MPI_INT,procid+sqrt_numprocs
,FOURTH,MPI_COMM_WORLD,&status);
                //Sends its up to up process
                if(row_index==0)
                    MPI_Send(&input[0],length,MPI_INT,procid-
sqrt_numprocs+numprocs,FOURTH,MPI_COMM_WORLD);
                else
                    MPI_Send(&input[0],length,MPI_INT,procid-
sqrt_numprocs,FOURTH,MPI_COMM_WORLD);
            }
            //Cross Communication
            if(row_index % 2){//Odd Rows(Indexes starts from zero)
                //Receives data from top left
                if(procid%sqrt_numprocs==1)
                    ierr = MPI_Recv(&top_left,1,MPI_INT,procid-
1,FIFTH,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&top_left,1,MPI_INT,procid-
sqrt_numprocs-1,FIFTH,MPI_COMM_WORLD,&status);
                //Sends data to bottom right
                if(row_index==sqrt_numprocs-1){
```

```
                    if(procid%sqrt_numprocs==0)
                        MPI_Send(&input[length-1][length-
1],1,MPI_INT,1,FIFTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[length-1][length-
1],1,MPI_INT,procid+sqrt_numprocs+1-numprocs,FIFTH,MPI_COMM_WORLD);
                }else{
                    if(procid%sqrt_numprocs==0)
                        MPI_Send(&input[length-1][length-
1],1,MPI_INT,procid+1,FIFTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[length-1][length-
1],1,MPI_INT,procid+sqrt_numprocs+1,FIFTH,MPI_COMM_WORLD);
                }
                //Receives data from top right
                if(procid%sqrt_numprocs==0)
                    ierr = MPI_Recv(&top_right,1,MPI_INT,procid-
2*sqrt_numprocs+1,SIXTH,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&top_right,1,MPI_INT,procid-
sqrt_numprocs+1,SIXTH,MPI_COMM_WORLD,&status);
                //Sends data to bottom left
                if(row_index==sqrt_numprocs-1){
                    if(procid%sqrt_numprocs==1)
                        MPI_Send(&input[length-
1][0],1,MPI_INT,sqrt_numprocs,SIXTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[length-1][0],1,MPI_INT,procid-
numprocs+sqrt_numprocs-1,SIXTH,MPI_COMM_WORLD);
                }else{
                    if(procid%sqrt_numprocs==1)
                        MPI_Send(&input[length-
1][0],1,MPI_INT,procid+2*sqrt_numprocs-1,SIXTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[length-
1][0],1,MPI_INT,procid+sqrt_numprocs-1,SIXTH,MPI_COMM_WORLD);
                }
                //Send data to top right
                if(procid%sqrt_numprocs==0)
                    MPI_Send(&input[0][length-1],1,MPI_INT,procid-
2*sqrt_numprocs+1,SEVENTH,MPI_COMM_WORLD);
                else
                    MPI_Send(&input[0][length-1],1,MPI_INT,procid-
sqrt_numprocs+1,SEVENTH,MPI_COMM_WORLD);
                //Receives data from bottom left
                if(row_index==sqrt_numprocs-1){
                    if(procid%sqrt_numprocs==1)
                        MPI_Recv(&bottom_left,1,MPI_INT,sqrt_numprocs,SEVEN
TH,MPI_COMM_WORLD,&status);
```

```
                    else
                        MPI_Recv(&bottom_left,1,MPI_INT,procid-
numprocs+sqrt_numprocs-1,SEVENTH,MPI_COMM_WORLD,&status);
                }else{
                    if(procid%sqrt_numprocs==1)
                        MPI_Recv(&bottom_left,1,MPI_INT,procid+2*sqrt_numpr
ocs-1,SEVENTH,MPI_COMM_WORLD,&status);
                    else
                        MPI_Recv(&bottom_left,1,MPI_INT,procid+sqrt_numproc
s-1,SEVENTH,MPI_COMM_WORLD,&status);
                }
                //Receives data from bottom right
                if(row_index==sqrt_numprocs-1){
                    if(procid%sqrt_numprocs==0)
                        ierr = MPI_Recv(&bottom_right,1,MPI_INT,1,EIGHTH,MP
I_COMM_WORLD,&status);
                    else
                        ierr = MPI_Recv(&bottom_right,1,MPI_INT,procid+sqrt
_numprocs+1-numprocs,EIGHTH,MPI_COMM_WORLD,&status);
                }else{
                    if(procid%sqrt_numprocs==0)
                        ierr = MPI_Recv(&bottom_right,1,MPI_INT,procid+1,EI
GHTH,MPI_COMM_WORLD,&status);
                    else
                        ierr = MPI_Recv(&bottom_right,1,MPI_INT,procid+sqrt
_numprocs+1,EIGHTH,MPI_COMM_WORLD,&status);
                }
                //Sends data to top left
                if(procid%sqrt_numprocs==1)
                    MPI_Send(&input[0][0],1,MPI_INT,procid-
1,EIGHTH,MPI_COMM_WORLD);
                else
                    MPI_Send(&input[0][0],1,MPI_INT,procid-sqrt_numprocs-
1,EIGHTH,MPI_COMM_WORLD);
            }else{//Even Rows
                //Sends data to bottom right
                if(procid%sqrt_numprocs==0)
                    MPI_Send(&input[length-1][length-
1],1,MPI_INT,procid+1,FIFTH,MPI_COMM_WORLD);
                else
                    MPI_Send(&input[length-1][length-
1],1,MPI_INT,procid+sqrt_numprocs+1,FIFTH,MPI_COMM_WORLD);
                //Receives data from top left
                if(row_index==0){
                    if(procid%sqrt_numprocs==1)
                        ierr = MPI_Recv(&top_left,1,MPI_INT,numprocs,FIFTH,
MPI_COMM_WORLD,&status);
                    else
```

```
                        ierr = MPI_Recv(&top_left,1,MPI_INT,procid-
sqrt_numprocs-1+numprocs,FIFTH,MPI_COMM_WORLD,&status);
                }else{
                    if(procid%sqrt_numprocs==1)
                        ierr = MPI_Recv(&top_left,1,MPI_INT,procid-
1,FIFTH,MPI_COMM_WORLD,&status);
                    else
                        ierr = MPI_Recv(&top_left,1,MPI_INT,procid-
sqrt_numprocs-1,FIFTH,MPI_COMM_WORLD,&status);
                }
                //Sends data to bottom left
                if(procid%sqrt_numprocs==1)
                    MPI_Send(&input[length-1][0],1,MPI_INT,procid-
1+2*sqrt_numprocs,SIXTH,MPI_COMM_WORLD);
                else
                    MPI_Send(&input[length-
1][0],1,MPI_INT,procid+sqrt_numprocs-1,SIXTH,MPI_COMM_WORLD);
                //Receives data from top right
                if(row_index==0){
                    if(procid%sqrt_numprocs==0)
                        ierr = MPI_Recv(&top_right,1,MPI_INT,numprocs-
sqrt_numprocs+1,SIXTH,MPI_COMM_WORLD,&status);
                    else
                        ierr = MPI_Recv(&top_right,1,MPI_INT,procid-
sqrt_numprocs+1+numprocs,SIXTH,MPI_COMM_WORLD,&status);
                }else{
                    if(procid%sqrt_numprocs==0)
                        ierr = MPI_Recv(&top_right,1,MPI_INT,procid-
2*sqrt_numprocs+1,SIXTH,MPI_COMM_WORLD,&status);
                    else
                        ierr = MPI_Recv(&top_right,1,MPI_INT,procid-
sqrt_numprocs+1,SIXTH,MPI_COMM_WORLD,&status);
                }
                //Receives data from bottom left
                if(procid%sqrt_numprocs==1)
                    ierr = MPI_Recv(&bottom_left,1,MPI_INT,procid-
1+2*sqrt_numprocs,SEVENTH,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&bottom_left,1,MPI_INT,procid-
1+sqrt_numprocs,SEVENTH,MPI_COMM_WORLD,&status);
                //Sends data to top right
                if(row_index==0){
                    if(procid%sqrt_numprocs==0)
                        MPI_Send(&input[0][length-1],1,MPI_INT,numprocs-
sqrt_numprocs+1,SEVENTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[0][length-1],1,MPI_INT,procid-
sqrt_numprocs+1+numprocs,SEVENTH,MPI_COMM_WORLD);
                }else{
```

```
                    if(procid%sqrt_numprocs==0)
                        MPI_Send(&input[0][length-1],1,MPI_INT,procid-
2*sqrt_numprocs+1,SEVENTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[0][length-1],1,MPI_INT,procid-
sqrt_numprocs+1,SEVENTH,MPI_COMM_WORLD);
                }
                //Sends data to top left
                if(row_index==0){
                    if(procid%sqrt_numprocs==1)
                        MPI_Send(&input[0][0],1,MPI_INT,numprocs,EIGHTH,MPI
_COMM_WORLD);
                    else
                        MPI_Send(&input[0][0],1,MPI_INT,procid-
sqrt_numprocs-1+numprocs,EIGHTH,MPI_COMM_WORLD);
                }else{
                    if(procid%sqrt_numprocs==1)
                        MPI_Send(&input[0][0],1,MPI_INT,procid-
1,EIGHTH,MPI_COMM_WORLD);
                    else
                        MPI_Send(&input[0][0],1,MPI_INT,procid-
sqrt_numprocs-1,EIGHTH,MPI_COMM_WORLD);
                }
                //Receives data from bottom right
                if(procid%sqrt_numprocs==0)
                    ierr = MPI_Recv(&bottom_right,1,MPI_INT,procid+1,EIGHTH
,MPI_COMM_WORLD,&status);
                else
                    ierr = MPI_Recv(&bottom_right,1,MPI_INT,procid+sqrt_num
procs+1,EIGHTH,MPI_COMM_WORLD,&status);
            }
            //Combines its partition and data comes from communication
            int full_input[length+2][length+2];
            combine_matrices(length,input,full_input,top,bottom,right,left,
top_right,top_left,bottom_right,bottom_left);
            // Calculates the total value of neighbors of all entries and f
inds their next state
            compute_next_state(length,input,full_input);
        }
        //At the end of he all time steps sends data to master process
        MPI_Send(&input,length*length,MPI_INT,MASTER_PROCESS,END_MESSAGE,MP
I_COMM_WORLD);
    }
    ierr = MPI_Finalize();
}
```