CMPE300
ANALYSIS OF ALGORITHMS

STUDENT NAME: CİHAT KAPUSUZ
STUDENT ID: 2016400126

INSTRUCTOR: TUNGA GÜNGOR
TEACHING ASISTANT: M. UTKAN GEZER

PROJECT TITLE: PARALLEL GAME OF LIFE
PROJECT TYPE: PROGRAMMING PROJECT

SUBMISSION DATE: 22.12.2019

– Introduction

The problem was defined as the well-known game ‘Game of Life’. 1 refers to be alive and 0 refers to be dead. Each cell decides to be born, live or die according to its total number of 8 neighbours’ living status. The main aspect of the project is cellular automata. There is 1 manager processor and it divides the given task into worker processors. The worker processors can communicate to do necessary operations. In that way parallelization is acquired.1Also the user can define number of turns which is defined as [T]. After [T] turns workers send their parts of the task back to the manager. Manager processor gathers information from the worker processors and produced 1 output fort he given task.

– Program Interface

First, user should be sure his/her computer has C and C++ compiler, to do this aspect, he/she should run:

sudo apt install gcc g++ wget make

command through the terminal. After being sure the downloading of necessary interfaces can be done without a doubt.

Since project is an MPI project, it uses the MPI itself. To use the MPI, user should download .tar.gz file from

wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.2.tar.gz

After that user should run:

gunzip -c openmpi-4.0.2.tar.gz | tar xf –

to exract the archive. After the exraction, openmpi-4.0.2 folder will be exracted to that location by default.

Then run:

sudo ./configure --prefix=/usr/local

After that run:

sudo make all install

This will be sufficient for installation of MPI that is needed for the project.

If user encounters a problem such that “Error while loading libopen-rto.so” he/she should run:

sudo ldconfig

After that it will not create any other additional problem.

Since the interface is installed correctly, we can continue with ‘Program Execution’ part.

Since the project code is written in C++ language, it should be compiled in mpic++ to produce runnable game.

For a C++ source code like game_of_life.cpp, run

mpic++ game_of_life.cpp -o game

A compiled MPI executable ‘game’, can be run with this command

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

Where [M] is the number of processes to run ‘game’ on. If user wants n worker processes, there is 1 additional process so [M] should be n+1 to fulfill the requested number of workers. –oversubscribe flag allows to use logical cores rather than limiting the user with physical cores. input.txt, output.txt and [T] are given to the ‘game’ through command line arguments. The input.txt will contain 360 rows and 360 columns. [T] is the number of turns that the game will be played. output.txt is the output of the game.

− Program Execution

User inputs 360x360 matrix as an input while which is full of 1 and 0’s(binary), and inputs where the output is stored and the number of turns [T].

‘Game of Life’ has some basic rules:

1. If alive cell has less than 2 neighbours, it dies. (1 turns into 0, loneliness)

2. Else if alive cell has more than 3 neighbours, it dies. (1 turns into 0, overpopulation)

3. Else if dead cell has exactly 3 neighbours, it borns. (0 turns into 1)

4. Otherwise, it remains alive or dead. (keeps current state)

Game is periodic which means the edges and the corners of matrix have 8 neighbours, too. Top edge traits bottom edge as additional upper row and bottom edge traits top edge as additional lower row. Right edge traits left edge as additional right column and left edge traits right edge as additional left column.

Top left corner cell traits bottom right corner cell as its’ top left corner neighbour and bottom right corner cell traits top left corner cell as

its’ bottom right corner neighbour. Top right corner cell traits bottom left corner cell as its’ top right corner neighbour and bottom left corner cell traits top right corner cell as its’ bottom left corner neighbour.

− Input and Output

The input file: argv[1]
The output file: argv[2]
Number of turns[T]: argv[3]
Each element in a row are seperated with a “ “ and each row is seperated with “\n” in input. The output file is created with the same aspect. [T] defines the number of turns that will be played.

− Program Structure

-Splitting the data

The manager process’ rank is 0 and the other processes are workers. First the manager process reads the input file, then divides the received matrix into squares and sends messages to the corresponding ranks. For instance, if we have 5 processes (1 manager and 4 workers), manager sends the [0-179]x[0-179] part of the matrix to the process which has rank 1, the [0-179]x[180-359] part of the matrix to the process which has rank 2, the [180-359]x[0-179] part of the matrix to the process which has rank 3, the [180-359]x[180-359] part of the matrix to the process which has rank 4. Since all matrix is partitioned and sent to the corresponding ranks, worker processes can take actions.

-Communication between the workers

To send data between workers we should simply define which process receives the message and which process sends. To prevent from deadlocks, we should send the messages in order and receive them in the reverse order. To be simpler, let me define the number of processes as 17 (1 manager, 16 workers). So, the processes divides map into 16 equal squares and made up a 4x4 matrix. While process which has rank 6 sends in 4 directions (left, right, top, bottom), corresponding neighbours 5, 7, 2, 10 should receive the message, so they cannot call send before receiving. We can divide the processes into 2 partitions with this condition. While first group sends arrays, second group receives the

message and then send the message to the processes in the first group. If we continue with the example, processes with rank 1, 3, 6, 8, 9, 11, 13 results first group, and the rest of the worker processes results the second group. To conclude with general solution for this problem, even numbered rows' odd ranked processes and odd numbered rows' even ranked processes results first group which I stored in '**vector<int> ranker1**' for determining this.

To send corners we need another 2 partitions to avoid deadlocks. To define these 2 partitions, we can simply say even rows as first group and odd rows as second group. I stored the processes at the even rows int '**vector<int> ranker2**' While 1, 2, 3, 4 , 9, 10, 11, 12 send their corners, the other processes receive first and then send their corners.

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

Since the world is periodic, we define neighbour processes for the processes on the edges with respect to this table (a toroid).

-Why do we need the additional edges and corners?

To play 'Game of Life' each cell needs to know all of its neighbours. Since each worker process has their own nxn matrix. A cell which is at the top right corner at the nxn matrix can access only 3 of its neighbours without communicating with neighbour processes. To access its remaining 5 neighbours, it should take the bottom left corner ('**sbottomleftcorner**') of the top right neighbour process as top right corner neighbour('**rtoprightcorner**'), it should take the remaining 2 right neighbour cells from the right neighbour process, for this aspect it should take leftmost column ('**sleft**') of the right neighbour process as additional right column ('**rright**') and take the necessary parts of this array. it does the same logical thing for the remaining 2 top neighbours. Since I explained how to fulfill 8 neighbours need for the top right corner, the remaining corners are implemented with this approach.

There are edges on the nxn matrix, a cell on these edges can reach 5 of its neighbours only. The remaining neighbour should be sent from the corresponding neighbour process. For instance, we consider a cell which is at the left edge of the nxn matrix but not at the corner and call this **X**, **X** cannot reach its 3 neighbours which are on the left normally at the grid. To reach these neighbours the process should receive the rightmost column from the left neighbour process (the left neighbour process should send its rightmost column

(‘**sright**’)) as additional left column (‘**rleft**’). Suppose **X** is the ith cell of the left edge, after this communication **X** can reach its remaining 3 neighbour with **rleft[i-1]**, **rleft[i]** and **rleft[i+1]**. We can do the same for the remaining edges.

-Turn logic and gameplay

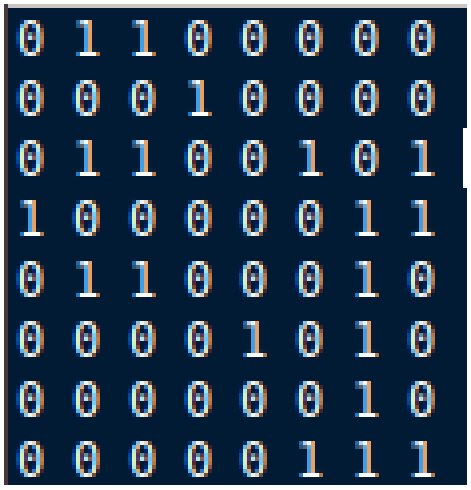
The turn [T] is given as input from user in argv[3], to determine how many turns the game is played. I simply defined **int turn** to know which turn the game is at and stored the argv[3] at **int turn_no** and put the communication of the processes and the game into a while loop which will continue until **turn** is equal to the **no_turn**. After that, each worker processes will send back their matrix and the manager process will put the messages into their appropriate locations. Then writes the main updated matrix into the output file.

Since we know how to reach all of the neighbours for a cell, we can apply the rules of the game. I stored the number of alive neighbours for a specific cell into an **int no_alive_neighbours**. After that, if that cell is alive(1) and it has more than 3 neighbours (**no_alive_neighbours>3**), cell dies(turns to 0) or it has less than 2 neighbours (**no_alive_neighbours<2**), cell dies(turns to 0). If the cell is not alive (0) and has exactly 3 neighbours (**no_alive_neighbours==3**), cell borns (turns to 1).

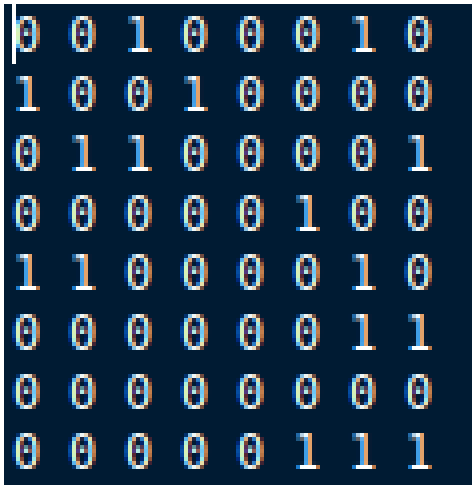
– Examples

A proper sample output should be 360x360 matrix, however, to be more understandable I used a screen output of 8x8 matrix.

Sample Input1:



Sample Output1:

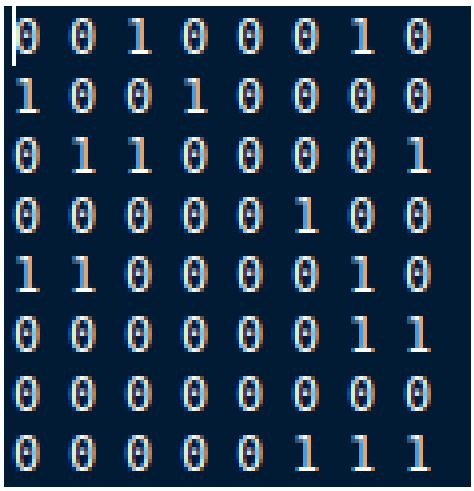


At the first row second cell dies because of loneliness. Number of alive neighbours is 1 which is less than 2, it dies and turns into 0 as it can be seen on the output with 1 turn.

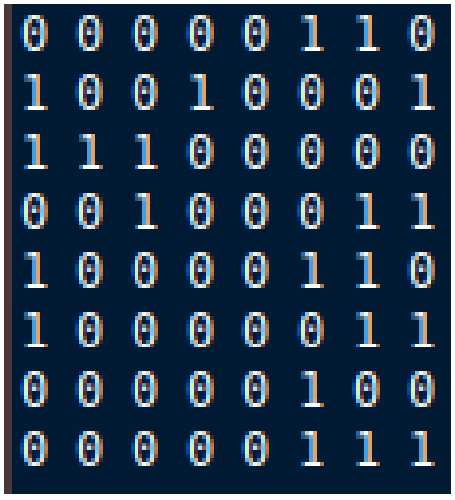
At the fourth row and eighth cell dies because of overpopulation. Number of alive neightbours is 4 which is more than 3, it dies and turns into 0 as it can be seen on the output with 1 turn.

At the first row seventh cell borns because it has exacly 3 neighbours. The cell turns into 1 as it can be seen on the output with 1 turn.

Sample Input2:



Sample Output2:



At the first row third cell dies because of loneliness. Number of alive neighbours is 1 which is less than 2, it dies and turns into 0 as it can be seen on the output with 1 turn.

At the third row first cell borns because it has exacly 3 neighbours. The cell turns into 1 as it can be seen on the output with 1 turn.

– Improvements and Extensions

Projects requirements are done perfectly, however there exists redundancy. We can get rid of redundancy with sending and receiving through functions (I improved my code in that aspect and created functions and get rid of redundant code). In that way there exists only one if/else if/else block instead of two if/else if/else blocks.

A possible improvement can be graphical user interface where the user gives input from an application not the terminal itself. However, this part of the project is not requested and not in the topic of the lesson, it is not necessary.

− Difficulties Encountered

I encountered so many deadlocks when I first implemented the project. The redundancy in the code caused a little bit trouble to see where are the deadlocks. After a while I debugged the code line by line and saw the misplaced order of send and receive calls.

Calculation of number of alive neighbours of the edges and the corners was difficult, because I need to decide which recieved part is necessary to calculate number of alive neighbours, going over and over again on my calculations helped me get rid of this problem.

The project is said to be much easier with python. However, I do not feel comfortable with python, so I implemented in C++.

Checkered implementation made it difficult because it is hard to handle 8 way send and receive without deadlock.

− Conclusion

The aim of the project acquired. The additional tasks which are periodic and checkered are also implemented and tested with the test cases. There are not any conflicts with the test cases so the code works properly even with bonus tasks.

Each worker processes only has the necessary information to play the game, no additional cell knowledge is given to the processes. They also get their part from the manager process and return back after the given turns are reached.

This project made me think to do multitasking for dividing labors to the worker processes which will help me a lot for the problems that I am already facing.

− Appendices

Source Code:

```
// Name: Cihat Kapusuz
// Student ID: 2016400126
// Compilation Status: Compiling
// Working Status: Working
// Periodic Bonus: Done
// Checkered Bonus: Done
#include "mpi.h"
#include <iostream>
```

```
#include <fstream>
#include <math.h>
#include <vector>
#include <algorithm>
#define MATRIX_SIZE 360
using namespace std;
void send_left_right_top_bot(int rank, int tag, int len_row, int size, int jump, int sleft[], int sright[], int stop[], int sbottom[]){
		// sends right edge to the right worker
		if((rank%jump)==0){ // if the processor on the right edge it sends the processor which has (jump-1) rank lower
			MPI_Send(sright, len_row, MPI_INT, (rank-jump+1), tag, MPI_COMM_WORLD);
		}
		else{
			MPI_Send(sright, len_row, MPI_INT, (rank+1), tag, MPI_COMM_WORLD);
		}
		// sends left edge to the left worker
		if((rank%jump)==1){ // if the processor on the left edge it sends the processor which has (jump-1) rank higher
			MPI_Send(sleft, len_row, MPI_INT, (rank+jump-1), tag, MPI_COMM_WORLD);
		}
		else{
			MPI_Send(sleft, len_row, MPI_INT, (rank-1), tag, MPI_COMM_WORLD);
		}
		// sends top edge to the top worker
		if(rank==jump){ // if the processor is at the right top corner it sends to highest ranked processor
			MPI_Send(stop, len_row, MPI_INT, (size-1), tag, MPI_COMM_WORLD);

		}
		else{
			MPI_Send(stop, len_row, MPI_INT, (rank+size-1-jump)%(size-1), tag, MPI_COMM_WORLD);
		}
		// sends bottom edge to the lower worker
		if(rank==(size-1-jump)){ // if the processor is at the 1 row upper to the right bottom corner it sends to highest ranked processor
			MPI_Send(sbottom, len_row, MPI_INT, (rank+jump), tag, MPI_COMM_WORLD);
		}
		else{
			MPI_Send(sbottom, len_row, MPI_INT, ((rank+jump)%(size-1)), tag, MPI_COMM_WORLD);
		}
}
void receive_right_left_bot_top(int rank, int tag, int len_row, int size, int jump, int rright[], int rleft[], int rbottom[], int rtop[]){
```

```
		// receives right edge of the right worker as additional left column
		if((rank%jump)==1){ // if the processor is at the left edge it will receive it's left column from (jump-1) rank higher
			MPI_Recv(rleft, len_row, MPI_INT, (rank+jump-1), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		else{
			MPI_Recv(rleft, len_row, MPI_INT, (rank-1), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		// receives left edge of the right worker as additional right column
		if((rank%jump)==0){ // if the processor is at the right edge it will receive it's left column from (jump-1) rank lower
			MPI_Recv(rright, len_row, MPI_INT, (rank-jump+1), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		else{
			MPI_Recv(rright, len_row, MPI_INT, (rank+1), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		// receives top edge of the bottom worker as additional bottom row
		if(rank==(size-1-jump)){ // if the processor is at the 1 row upper to the right bottom corner it receives from the highest ranked processor
			MPI_Recv(rbottom, len_row, MPI_INT, (rank+jump), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		else{
			MPI_Recv(rbottom, len_row, MPI_INT, (rank+jump)%(size-1), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		// receives bottom edge of the top worker as additional top row
		if(rank==jump){ // if the processor is at the right top corner it recieves from the highest ranked processor
			MPI_Recv(rtop, len_row, MPI_INT, size-1, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		else{
			MPI_Recv(rtop, len_row, MPI_INT, (rank+size-1-jump)%(size-1), tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
}
void send_all_corners(int rank, int tag, int size, int jump, int stopleftcorner[], int stoprightcorner[], int sbottomleftcorner[], int sbottomrightcorner[]){
//sendtopleft
		if(rank==1){ // if rank is 1, send top left corner to the highest ranked processor
			MPI_Send(stopleftcorner, 1, MPI_INT, (size-1), tag, MPI_COMM_WORLD);
		}
		else if(rank%jump==1){ // else if processor is at left edge, send top left corner to previous processor
```

```
                MPI_Send(stopleftcorner, 1, MPI_INT, (rank-1), tag,
MPI_COMM_WORLD);
        }
        else if(rank<=jump){ // else if processor is at top edge, send top left corner to (size-1-
jump-1) process higher
                MPI_Send(stopleftcorner, 1, MPI_INT, (rank-jump-1+size-1), tag,
MPI_COMM_WORLD);
        }
        else{
                MPI_Send(stopleftcorner, 1, MPI_INT, (rank-jump-1), tag,
MPI_COMM_WORLD);
        }
        //sendtopright
        if(rank==jump){ // if rank equals jump, send top right corner to the processor which is
at bottom left
                MPI_Send(stoprightcorner, 1, MPI_INT, (size-1-jump+1), tag,
MPI_COMM_WORLD);
        }
        else if(rank%jump==0){ // else if processor is at right edge, send top right corner to
the processor which is ((2*jump)-1) rank lower
                MPI_Send(stoprightcorner, 1, MPI_INT, (rank-(2*jump)+1), tag,
MPI_COMM_WORLD);
        }
        else if(rank<=jump){ // else if processor is at top edge, send top right corner to (size-
1-jump+1) processor higher
                MPI_Send(stoprightcorner, 1, MPI_INT, (rank-jump+1+size-1), tag,
MPI_COMM_WORLD);
        }
        else{
                MPI_Send(stoprightcorner, 1, MPI_INT, (rank-jump+1), tag,
MPI_COMM_WORLD);
        }
        //sendbottomleft
        if(rank==(size-1-jump+1)){ // if the processor is at the bottom left corner, send bottom
left corner to the processor which is at top right corner
                MPI_Send(sbottomleftcorner, 1, MPI_INT, (jump), tag,
MPI_COMM_WORLD);
        }
        else if(rank%jump==1){ // else if the processor is at the left edge, send bottom left
corner to the processor which is ((2*jump)-1) rank higher
                MPI_Send(sbottomleftcorner, 1, MPI_INT, (rank+(2*jump)-1), tag,
MPI_COMM_WORLD);
        }
        else if(rank>=(size-1-jump+1)){ // else if processor is at bottom edge, send bottom left
corner to the processor which is (size-1-jump+1) rank lower
                MPI_Send(sbottomleftcorner, 1, MPI_INT, (jump+rank-size), tag,
MPI_COMM_WORLD);
        }
        else{
```

```
                MPI_Send(sbottomleftcorner, 1, MPI_INT, (rank+jump-1), tag,
MPI_COMM_WORLD);
        }
        //sendbottomright
        if(rank==(size-1)){ // if the processor is at bottom right corner, send bottom right
corner to the first processor
                MPI_Send(sbottomrightcorner, 1, MPI_INT, (1), tag,
MPI_COMM_WORLD);
        }
        else if(rank%jump==0){ // else if the processor is at the right edge, send bottom right
corner to the next processor
                MPI_Send(sbottomrightcorner, 1, MPI_INT, (rank+1), tag,
MPI_COMM_WORLD);
        }
        else if(rank>=(size-1-jump+1)){ // else if the processor is at the bottom edge, send
bottom right corner to the processor which is (size-1-jump-1) rank lower
                MPI_Send(sbottomrightcorner, 1, MPI_INT, (rank+jump+1-(size-1)), tag,
MPI_COMM_WORLD);
        }
        else{
                MPI_Send(sbottomrightcorner, 1, MPI_INT, (rank+jump+1), tag,
MPI_COMM_WORLD);
        }
}
void receive_all_corners(int rank, int tag, int size, int jump, int rbottomrightcorner[], int
rbottomleftcorner[], int rtoprightcorner[], int rtopleftcorner[]){
        //recievebottomright
        if(rank==(size-1)){ // if processor is at right bottom corner, receive bottom right
corner from processor which has rank 1
                MPI_Recv(rbottomrightcorner, 1, MPI_INT, (1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank%jump==0){ // else if processor is at right edge, receive bottom right
corner from next processor
                MPI_Recv(rbottomrightcorner, 1, MPI_INT, (rank+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank>=(size-1-jump+1)){ // else if processor is at bottom edge, receive bottom
right corner from the processor which is (size-1-jump-1) rank lower
                MPI_Recv(rbottomrightcorner, 1, MPI_INT, (jump-size+1+rank+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else{
                MPI_Recv(rbottomrightcorner, 1, MPI_INT, (rank+jump+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        //receivebottomleft
        if(rank==(size-1-jump+1)){ // if processor is at left bottom corner, receive bottom left
corner from processor which has rank jump
```

```
                MPI_Recv(rbottomleftcorner, 1, MPI_INT, (jump), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank%jump==1){ // else if processor is at left edge, receive bottom left corner
from the processor which is ((2*jump)-1) rank higher
                MPI_Recv(rbottomleftcorner, 1, MPI_INT, (rank+(2*jump)-1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank>=(size-1-jump+1)){ // else if processor is at bottom edge, receive bottom
right corner from the processor which is ((size-1)-jump+1) rank lower
                MPI_Recv(rbottomleftcorner, 1, MPI_INT, (rank-(size-1)+jump-1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else{
                MPI_Recv(rbottomleftcorner, 1, MPI_INT, (rank+jump-1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        //receivetopright
        if(rank==jump){ // if processor is at right top corner, receive top right corner from
processor which is at left bottom corner
                MPI_Recv(rtoprightcorner, 1, MPI_INT, (size-1-jump+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank%jump==0){ // if processor is at right edge, receive top right corner from
processor which is ((2*jump)-1) rank lower
                MPI_Recv(rtoprightcorner, 1, MPI_INT, (rank-(2*jump)+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank<=jump){ // if process is at top edge, receive top right corner from
processor which is (size-1-jump+1) rank higher
                MPI_Recv(rtoprightcorner, 1, MPI_INT, (size-1+rank-jump+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else{
                MPI_Recv(rtoprightcorner, 1, MPI_INT, (rank-jump+1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        //receivetopleft
        if(rank==1){ // if processor is at left top corner, receive top left corner from processor
which is at right bottom corner
                MPI_Recv(rtopleftcorner, 1, MPI_INT, (size-1), tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }
        else if(rank%jump==1){ // else if processor is at left edge, receive top left corner from
previous processor
                MPI_Recv(rtopleftcorner, 1, MPI_INT, (rank-1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else if(rank<=jump){ // else if processor is at top edge, receive top left corner from
processor which is (size-1-jump-1) rank higher
```

```
                        MPI_Recv(rtopleftcorner, 1, MPI_INT, (rank+(size-1)-jump-1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
                else{
                        MPI_Recv(rtopleftcorner, 1, MPI_INT, (rank-jump-1), tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
}

int main(int argc, char *argv[])
{
   int rank, size, matrix[MATRIX_SIZE][MATRIX_SIZE], tag = 201; // defines rank, size main
matrix and tag
   int stoprightcorner[1], rtoprightcorner[1];                  // defines send and recieve top right
corner
   int stopleftcorner[1], rtopleftcorner[1];                    // defines send and recieve top left
corner
   int sbottomrightcorner[1], rbottomrightcorner[1];            // defines send and recieve bottom
right corner
   int sbottomleftcorner[1], rbottomleftcorner[1];          // defines send and recieve bottom left
corner

   MPI_Init(&argc, &argv); //Starts the MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // gets rank of the processor
   MPI_Comm_size(MPI_COMM_WORLD, &size); // gets how many processor are in the program
        int no_alive_neighbour=0;       // number of alive neighbour to compare with game
restrictions
        int jump= (int)sqrt((size-1));    // sqrt of the worker processes
        int len_row = MATRIX_SIZE/jump;   // number of rows and columns in 1 worker processor
        int message[len_row][len_row];    // part of array which will be stored in the processor
        int temp[len_row][len_row];       // temporary array to update each node with respect to
'Game of Life'
        int sright[len_row];            // an array that will send right edge of the belonging processor
        int sleft[len_row];                 // an array that will send left edge of the belonging processor
        int stop[len_row];                  // an array that will send top edge of the belonging processor
        int sbottom[len_row];           // an array that will send bottom edge of the belonging processor
        int rright[len_row];                // an array that will hold received additional right edge from
the left processor
        int rleft[len_row];                 // an array that will hold received additional left edge from
the right processor
        int rtop[len_row];                  // an array that will hold received additional top edge from
the top processor
        int rbottom[len_row];               // an array that will hold received additional bottom edge
from the bottom processor
        int no_turn=atoi(argv[3]);      // number of turns that game will be played
        int turn=0;                 // initialize current turn
        vector<int> ranker1;                // to decide which processors will send its top, bottom, left
and right edge first then recieve
        vector<int> ranker2;                // to decide which processors will send its top right-left and
bottom right-left corner first then recieve
```

```
	for(int i=0; i<jump; i+=2){
		for(int j=1; j<jump; j+=2){
			ranker1.push_back(i*jump+j); // adds odd ranked processors in even rows
		}
	}
	for(int i=0; i<jump; i+=2){
		for(int j=2; j<=jump; j+=2){
			ranker1.push_back((i+1)*jump+j);  // adds even ranked processors in odd
rows
		}
	}
	for(int i=0; i<jump; i+=2){
		for(int j=1; j<=jump; j++){
			ranker2.push_back(i*jump+j);  // adds even numbered rows' processors
		}
	}

   if (0 == rank) {
	//take input into the manager process
	ifstream myfile;
	myfile.open(argv[1]);
		if (myfile.is_open()) {
			for(int i=0; i<MATRIX_SIZE; i++){
				for(int j=0; j<MATRIX_SIZE; j++){
					myfile >> matrix[i][j];
				}
			}
		}
	myfile.close();
	//take input completed

	//split messages with corresponding ranks and send them
	for(int i=0; i<MATRIX_SIZE; i+=len_row){ // i represents the corresponding row's first index
		for( int j=0; j<MATRIX_SIZE; j+=len_row){ // j represents the corresponding
column's first index
			for(int k=i; k<i+len_row; k++){
				for(int l=j; l<j+len_row; l++){
					message[k-i][l-j]=matrix[k][l];
				}
			}
			MPI_Send(message, MATRIX_SIZE*MATRIX_SIZE/(jump*jump), MPI_INT,
(((i*jump*jump)/MATRIX_SIZE)+(j/(len_row)))+1, tag, MPI_COMM_WORLD);
		}
	}
	//message send from rank 0 is done

   }

	if (rank!=0) {
```

```
        MPI_Recv(message, MATRIX_SIZE*MATRIX_SIZE/(jump*jump), MPI_INT, 0, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE); // receive all messages that have sent from rank 0
            //start turn loop
            while(turn<no_turn){
            turn++;
            // initialize temp array to be used for 'Game of Life' restrictions
            for(int i=0; i<len_row; i++){
                for(int j=0; j<len_row; j++){
                    temp[i][j]=message[i][j];
                }
            }
            // initialization of temp array finished
            // initialize necessary parts that needs to be sent to other processes
          for(int i=0; i<len_row; i++){
                sright[i]=message[i][(len_row)-1];          // gets right part of the its grid
before send
                sleft[i]=message[i][0];                     // gets left part of the its grid
before send
                stop[i]=message[0][i];                      // gets top part of the its grid
before send
                sbottom[i]=message[(len_row)-1][i];         // gets bottom part of the its
grid before send
            }
            stopleftcorner[0]=message[0][0];            // gets top left corner of the message
before sending it
            stoprightcorner[0]=message[0][(len_row)-1];         // gets top right corner of
the message before sending it
            sbottomleftcorner[0]=message[(len_row)-1][0];       // gets bottom left corner of
the message before sending it
            sbottomrightcorner[0]=message[(len_row)-1][(len_row)-1];  // gets bottom right
corner of the message before sending it
        // sends and receives top, bottom, left and right edges
        if (find(ranker1.begin(), ranker1.end(), rank) != ranker1.end()){

            send_left_right_top_bot(rank, tag, len_row, size, jump, sleft, sright, stop, sbottom);
            receive_right_left_bot_top(rank, tag, len_row, size, jump, rright, rleft, rbottom, rtop);

        } else {
            receive_right_left_bot_top(rank, tag, len_row, size, jump, rright, rleft, rbottom, rtop);
            send_left_right_top_bot(rank, tag, len_row, size, jump, sleft, sright, stop, sbottom);
        }
        // sends and receives top, bottom, left and right edges are done

        // send and receives top left-right corners, bottom left-right corners
        if (find(ranker2.begin(), ranker2.end(), rank) != ranker2.end()){
            send_all_corners(rank, tag, size, jump, stopleftcorner, stoprightcorner,
sbottomleftcorner, sbottomrightcorner);
            receive_all_corners(rank, tag, size, jump, rbottomrightcorner, rbottomleftcorner,
rtoprightcorner, rtopleftcorner);
        }
```

```
	else{
		receive_all_corners(rank, tag, size, jump, rbottomrightcorner, rbottomleftcorner, rtoprightcorner, rtopleftcorner);
		send_all_corners(rank, tag, size, jump, stopleftcorner, stoprightcorner, sbottomleftcorner, sbottomrightcorner);
	}
	// all corner are sent and received

	// message passing done, implement the 'Game of Life'
	for(int i=0; i<len_row; i++){
		for(int j=0; j<len_row; j++){
			//implementcorners
			if((i==0) && (j==0)){ // calculate top left corner

	no_alive_neighbour=rtopleftcorner[0]+rleft[0]+rleft[1]+rtop[0]+rtop[1]+message[0][1]+message[1][0]+message[1][1];
			}else if((i==0) && (j==(len_row-1))){ // calculate top right corner

	no_alive_neighbour=rtoprightcorner[0]+rright[0]+rright[1]+rtop[(len_row-2)]+rtop[(len_row-1)]+message[0][(len_row-2)]+message[1][(len_row-1)]+message[1][(len_row-2)];
			}else if((i==(len_row-1)) && (j==(len_row-1))){ // calculate bottom right corner
				no_alive_neighbour=rbottomrightcorner[0]+rright[(len_row-1)]+rright[(len_row-2)]+rbottom[(len_row-2)]+rbottom[(len_row-1)]+message[(len_row-1)][(len_row-2)]+message[(len_row-2)][(len_row-1)]+message[(len_row-2)][(len_row-2)];
			}else if((i==(len_row-1)) && (j==0)){ // calculate bottom left corner
				no_alive_neighbour=rbottomleftcorner[0]+rleft[(len_row-1)]+rleft[(len_row-2)]+rbottom[0]+rbottom[1]+message[(len_row-1)][1]+message[(len_row-2)][0]+message[(len_row-2)][1];
			}//corners are done
			//implement the edges
			else if(i==0){ // calculate top edge without corners
				no_alive_neighbour=rtop[j-1]+rtop[j]+rtop[j+1]+message[0][j-1]+message[0][j+1]+message[1][j-1]+message[1][j]+message[1][j+1];
			}else if(j==0){ // calculate left edge without corners
				no_alive_neighbour=rleft[i-1]+rleft[i]+rleft[i+1]+message[i-1][0]+message[i+1][0]+message[i-1][1]+message[i][1]+message[i+1][1];
			}else if(j==(len_row-1)){ // calculate right edge without corners
				no_alive_neighbour=rright[i-1]+rright[i]+rright[i+1]+message[i-1][(len_row-1)]+message[i+1][(len_row-1)]+message[i-1][(len_row-2)]+message[i][(len_row-2)]+message[i+1][(len_row-2)];
			}else if(i==(len_row-1)){ // calculate bottom edge without corners
				no_alive_neighbour=rbottom[j-1]+rbottom[j]+rbottom[j+1]+message[(len_row-1)][j-1]+message[(len_row-1)][j+1]+message[(len_row-2)][j-1]+message[(len_row-2)][j]+message[(len_row-2)][j+1];
			}
			//edges are done
			else{
				no_alive_neighbour=message[i-1][j-1]+message[i-1][j]+message[i-1][j+1]+message[i][j-1]+message[i][j+1]+message[i+1][j-1]+message[i+1][j]+message[i+1][j+1];
```

```
                        }
                        // 'Game of Life' rules are applied
                        if((message[i][j]==0) && (no_alive_neighbour==3)){
                                temp[i][j]=1;
                                }
                        if((message[i][j]==1) && ((no_alive_neighbour<2) ||
(no_alive_neighbour>3))){
                                temp[i][j]=0;
                        }
                        // life or death has decided and stored in temp array
                }
        }
        // update the changes from the round
        for(int i=0; i<len_row; i++){
                for(int j=0; j<len_row; j++){
                        message[i][j]=temp[i][j];
                        }
        }
        // update finished
}
//turn loop finished
        MPI_Send(message, MATRIX_SIZE*MATRIX_SIZE/(jump*jump), MPI_INT, 0, tag,
MPI_COMM_WORLD); // send back grid back to the rank 0 process
        }
        if (0 == rank) {
        // manager recieves the matrix back from its workers and puts messages to appropriate
location
        for(int i=0; i<MATRIX_SIZE; i+=len_row){
                for( int j=0; j<MATRIX_SIZE; j+=len_row){
                MPI_Recv(message, MATRIX_SIZE*MATRIX_SIZE/(jump*jump), MPI_INT,
(((i*jump*jump)/MATRIX_SIZE)+(j/(len_row)))+1,tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE); //recieve message
                        for(int k=i; k<i+len_row; k++){
                                for(int l=j; l<j+len_row; l++){
                                        matrix[k][l]=message[k-i][l-j]; // update first matrix
                                }
                        }
                }
        }

        //write to output file
        ofstream outputfile;
        outputfile.open(argv[2]);
        for(int i=0; i<MATRIX_SIZE; i++){
                for(int j=0; j<MATRIX_SIZE; j++){
                outputfile << matrix[i][j] << " ";
                }
        outputfile << endl;
        }
        outputfile.close();
```

```
            }
            //output operation is done

    MPI_Finalize(); // finalize the mpi
    return 0;
}
```