CMPE300 MPI PROJECT

Course ID – 300

Course Name – Analysis of Algorithms

Student Name – Halil İbrahim Orhan

Type of the Project – Programming Project

Submission Date – 20.12.2019

1.)Introduction

In this project we are asked to implement Cellular Automata system. Our implementation focuses on “Game of Life” which consists of a map with cells and creatures. 1 means creature exists in the map and 0 means it doesn’t. Each creature has 8 neighbors around them. Rules are created according to neighbors. Rules are:

1.) Loneliness kills: A creature dies if it has less than 2 neighboring creatures, not counting itself.
2.) Overpopulation kills: A creature dies if it has more than 3 neighboring creatures.
3.) Reproduction: A new life appears on an empty cell if it has exactly 3 neighboring creatures.
4.) In any other condition, the creatures remain alive.

Map’s state is initialized at time t=0. After that to calculate map in the next time step, we should look at the previous time step map. As the rules explained above already next map is calculated according to neighbors of each cell. We keep on simulating like this until time t = T.

2.)Program Interface

In this project we are asked to implement our project using parallelism. We implement this parallelism with MPI(Message Passing Interface). This interface must be installed before the execution. Before program can be executed it must be compiled. Since code is written in c++ language, it can be compiled with this command:

mpic++ project.cpp -o game

Then we run our code with this command:

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

Parameters in this code is explained below:

[M] : This is the number of processes to run game on. There is manager process and worker processes(C). Since program is implemented using stripped split, number of process has to be given as C is an even number and divides 360

[T] : This is the number of iterations to simulate the Game.

input.txt : This is the file which contains the first map written in matrix form.

output.txt : This the file name that user wants to write in. Output file isn’t has to exists. Program creates the file.

User can wait for programs termination and see solution in output.txt or he can use ctrl +c and terminates the program before it completes its execution. User can't see the solution in output.txt.

3.)Program Execution

Main functionality of this program is to simulate a game. Program takes an input, which is format explained below, as an initial map. After that program starts to compute next map using previous map. Computation is done by the rule explained in the introduction section. For each element in the previous map program looks up to 8 neighbor of that element and states it as 1 or 0 in the next map. Program is implemented with Cutoff method which means in boundaries since they don't have neighbors in some edges they are taken as 0. This continues as long as t is equal to given time step. When computation finished it starts to write to output file.

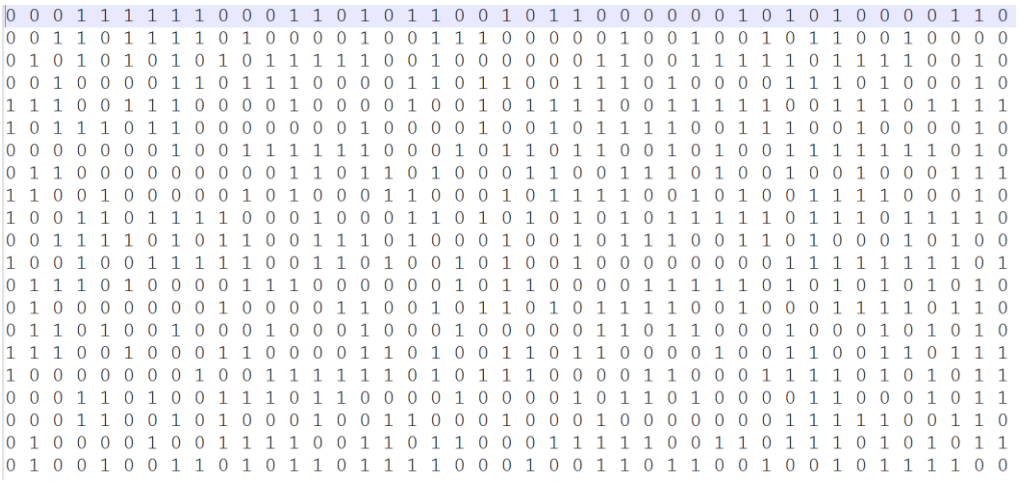
Input file must be a matrix 360 to 360. Matrix contains 1's or 0's. Input file must be in the same directory with the program. Output file doesn't have to exist in directory because if not program creates a file with the same name with given argument. If file exists it changes its contents with the programs output. More detailed explanation will be given in "Input and Output" section.

Number of iteration is also given as argument. If this number is zero or less than zero program takes it as zero and if it is 0 output will be same with the input file. Also as mentioned above number of processes have to given as number of worker processes is even and divides 360

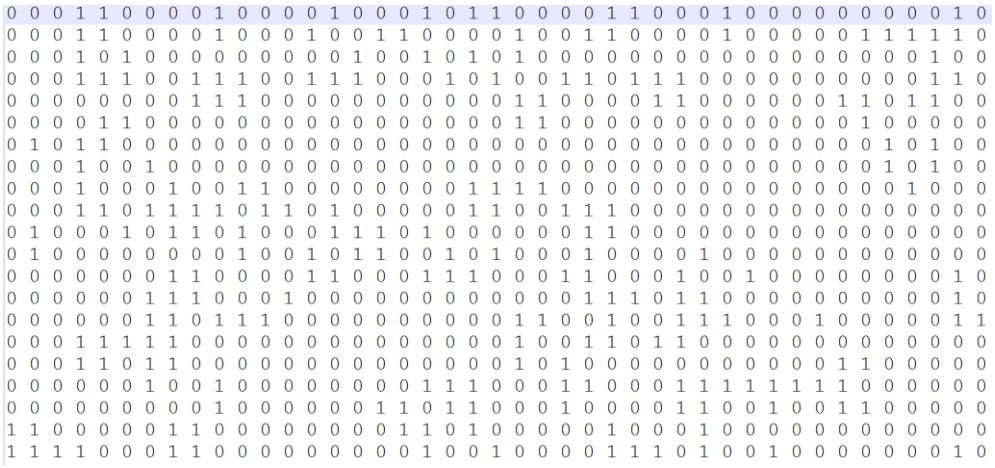
4.)Input and Output

Input has to be in a specified format. This format has to be in a matrix-like form which consists of 0's and 1's. Elements has to be written with space character between them. After 360 elements in a row, next row 360 elements have to be in the next line. There has to be 360 row and 360 column in the input file. Output file will be computed from input file. Therefore it will be in the same format with the input file.

Example input format:



Example output format after 5 time step:



5.) Program Structure

Input and MPI Initializations

Program starts with MPI initialization. This must be done for to use MPI interface. After that it takes the id of the current process with variable “curIdPs”. This will help to understand which process we are in. With that also total process number is taken. Variable totNumPs. These information’s are gathered from MPI functions MPI_Comm_rank() and MPI_Comm_size(). Input operations must be done in manager process that its rank is 0. After that a big if else block is used to separate worker processes from manager process. Program checks if its id is 0.

Arguments given in command line is taken and given to variables to use. After input file name is taken as variable it is used to take initial matrix from the file. Ifstream methods

are used to take input. 2D array used to put the map in input file to a variable. matrix[360][360] is used for input. Every element in input is read one by one and put in the matrix.

Manager and Workers Communication

After map is created, since I used Striped split method, map is sliced to equal number of rows and 360 column. rowNum is found by division of 360 to number of worker process. sendArray is used for sliced matrices. Every sliced matrix is sent with MPI_Send() function. All of the send matrices's source is process 0 and their destinations are workers ranks. Since it will be used program also sends rowNum.

Workers in the else block wait to receive information from the manager process with MPI_Recv(). Both methods of MPI are blocking. Workers do the calculations on their part of the matrix. While they do their operations manager waits for computed matrices with MPI_Recv() function. Manager concatenate matrix slices into lastArray[][]. This is the last form of the matrix.

Communication Between Workers

Communication between workers is provided with blocking MPI methods. All worker processes must communicate with the process above and below. Because we should look up for the top row process below because we should look up for below row of bottom row of split matrix. What happens for bottom row needed also for top row. So worker processes send each other their bottom and top row.

Our purpose is to implement this project with parallel programming. However, if these processes send their bottom and top row sequentially it would harm parallel programing perspective. There is a waiting chain in this scheme. This scheme works in $O(C \cdot \sqrt{n})$. To avoid that part of the processes send, other part of the processes receive. Every process with an odd rank sends its bottom row to process below. Then they wait to receive a row of information from the process above. It happens the same for sending top row and receiving bottom row. Every process with an even rank receives a row of information from the process above. Then, they send their bottom row to the process below. It is a way to achieve $O(\sqrt{n})$ time for the communications.

Computation In Worker Processes

Main purpose here is to calculate map in the next iteration. Next matrix is kept in tempArray[][]. For every element in split matrix program looks for 8 neighbors of that element. There is 8 method for each of them. In these methods boundaries are checked and their neighbors out of the maps are returned 0. For rest of them if there is 1 in checked neighbor method returns 1. In computation these methods are added. After that, rules explained in the introduction section are checked and according to that next value of that element is decided and put in the tempArray. Then sendArray is equalized to tempArray.

Operation in previous section "Communication Between Workers" and this section "Computation In Worker Processes" is done in a for loop. These operations done for given number of times.

Output

After all computations and end of iterations split matrix are sent back to manager process for output operations. Manager process receives split matrix with MPI_Recv() one by one starting from rank 1 to the end of all worker processes. Output is written to output file with ofstream. It is written to file one by one from received matrices. Also space character and new line is added to this file.

6.)Examples

0	1	1	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0

Time step= t Time step=t+1

In first example 3x3 matrix is used. This example shows loneliness kills and overpopulation kills

0	1	0	1	0	0
1	1	0	0	1	0
0	0	1	0	1	0

Time step=t Time step=t+1

This example shows reproduction of creature

7.)Improvements and Extensions

In my implementation I used Stripped Split and Cutoff methods. Using periodic method may be more generic and it could give more realistic results. There might be another improvement in manager receiving matrices from workers. Although we used even and odd number separation in workers communication to avoid sequential waiting, there is also a waiting sequence in manager process receiving matrices from workers. There might be an improvement in this part.

8.) Difficulties Encountered

One of the difficulties I faced in this project is deadlock problem. When two process try to send a message at the same time. But I couldn't figure out where this problem happened in code. Then I print something between al parts and I found out. Other problem I faced was not a difficult one but it took some time of me. I made some mistakes about which operations will be done in iteration for.

9.) Conclusion

In this project I implemented a game simulation with parallel programming using OpenMPI. I saw that only using a parallel programing interface is not enough. We should also help this parallelism with our implementation. So I have learnt how to implement this kind of programs.

10.) Appendix

```
//Halil İbrahim Orhan
//2016400054
//Compiling
//Working
#include <mpi.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
using namespace std;
int leftTop(int t,int k,int curIdPs,int sendArray[][360],int recTopRow[]){
	if(k==0){
		return 0;
	}else if(t==0){
		if(curIdPs==1){
```

```
                return 0;
        }else{
                if(recTopRow[k-1]==1){
                        return 1;
                }
                return 0;
        }
    }else{
        if(sendArray[t-1][k-1]==1){
                return 1;
        }
        return 0;
    }
}
int top(int t,int k,int curIdPs,int sendArray[][360],int recTopRow[]){
    if(t==0){
        if(curIdPs==1){
                return 0;
        }else{
                if(recTopRow[k]==1){
                        return 1;
                }
                return 0;
        }
    }else{
        if(sendArray[t-1][k]==1){
                return 1;
        }
        return 0;
```

```
	}
}
int rightTop(int t,int k,int curIdPs,int sendArray[][360],int recTopRow[]){
	if(k==359){
		return 0;
	}else if(t==0){
		if(curIdPs==1){
			return 0;
		}else{
			if(recTopRow[k+1]==1){
				return 1;
			}
			return 0;
		}
	}else{
		if(sendArray[t-1][k+1]==1){
			return 1;
		}
		return 0;
	}

}
int left(int t,int k,int curIdPs,int sendArray[][360],int recTopRow[]){
	if(k==0){
		return 0;
	}else{
		if(sendArray[t][k-1]==1){
			return 1;
		}
```

```
		return 0;
	}
}
int right(int t,int k,int curIdPs,int sendArray[][360],int recTopRow[]){
	if(k==359){
		return 0;
	}else{
		if(sendArray[t][k+1]==1){
			return 1;
		}
		return 0;
	}
}
int botLeft(int t,int k,int curIdPs,int sendArray[][360],int recBotRow[],int rowNum,int totNumPs){
	if(k==0){
		return 0;
	}else if(t==rowNum-1){
		if(curIdPs+1==totNumPs){
			return 0;
		}else{
			if(recBotRow[k-1]==1){
				return 1;
			}
			return 0;
		}
	}else{
		if(sendArray[t+1][k-1]==1){
			return 1;
		}
```

```
                return 0;
        }
}
int bot(int t,int k,int curIdPs,int sendArray[][360],int recBotRow[],int rowNum,int totNumPs){
        if(t==rowNum-1){
                if(curIdPs+1==totNumPs){
                        return 0;
                }else{
                        if(recBotRow[k]==1){
                                return 1;
                        }
                        return 0;
                }
        }else{
                if(sendArray[t+1][k]==1){
                        return 1;
                }
                return 0;
        }
}
int botRight(int t,int k,int curIdPs,int sendArray[][360],int recBotRow[],int rowNum,int totNumPs){
        if(k==359){
                return 0;
        }else if(t==rowNum-1){
                if(curIdPs+1==totNumPs){
                        return 0;
                }else{
                        if(recBotRow[k+1]==1){
                                return 1;
```

```
                }
                return 0;
            }
        }else{
            if(sendArray[t+1][k+1]==1){
                return 1;
            }
            return 0;
        }
}
int main(int argc, char*argv[])
{
        int iteration=atoi(argv[3]);
        MPI_Init(NULL,NULL);
        int curIdPs;//id of current process
        MPI_Comm_rank(MPI_COMM_WORLD,&curIdPs);
        int totNumPs;//total number of process
        MPI_Comm_size(MPI_COMM_WORLD,&totNumPs);
        if(curIdPs==0){
            string output=argv[2];

            ifstream myfile(argv[1]);
            if(!myfile.is_open()){
                cout<<"file error"<< endl;
            }
            int matrix[360][360]; //input matrix
            int i=0;
            int j=0;
            string line;
```

```
        while(getline(myfile,line)){
                stringstream s(line);
                int x=0;
                j=0;
                while(s>>x){
                        matrix[i][j]=x;
                        j++;

                }
                i++;
        }
        int counter=0;
        int rankNum=1;
        int rowNum=360/(totNumPs-1);
        int sendArray[rowNum][360]; // matrix parts that will be send

        for(int t=0;t<360;t++){
                for(int k=0;k<360;k++){
                        sendArray[counter][k]=matrix[t][k];

                }

                if(counter+1==rowNum){

MPI_Send(&rowNum,1,MPI_INT,rankNum,0,MPI_COMM_WORLD);

MPI_Send(&sendArray,360*rowNum,MPI_INT,rankNum,1,MPI_COMM_WORLD);
                        rankNum++;
                        counter=0;
```

```
                }else{
                        counter++;
                }

        }


        ofstream outfile;
        counter=0;
        outfile.open(output);
        for(int t=1;t<totNumPs;t++){
                int lastArray[rowNum][360];//recieved matrix parts

    MPI_Recv(&lastArray,360*rowNum,MPI_INT,t,0,MPI_COMM_WORLD,MPI_STATUS_I
GNORE);
                for(int k=0;k<rowNum;k++){
                        for(int l=0;l<360;l++){
                                outfile<<lastArray[k][l]<<" ";
                        }
                        outfile<<endl;
                }
        }
        outfile.close();


    }else{
        int sendTopRow[360];
        int sendBotRow[360];
        int recTopRow[360];// recieve the row from process above
        int recBotRow[360];// recieve the row from process below
        int rowNum;
```

```
MPI_Recv(&rowNum,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        int sendArray[rowNum][360];

MPI_Recv(&sendArray,360*rowNum,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        for(int i=0;i<iteration;i++){
            for(int j=0;j<360;j++){
                sendBotRow[j]=sendArray[rowNum-1][j];
                sendTopRow[j]=sendArray[0][j];
            }
            if(curIdPs%2==1){
                if(curIdPs==1){

MPI_Send(&sendBotRow,360,MPI_INT,2,0,MPI_COMM_WORLD);

MPI_Recv(&recBotRow,360,MPI_INT,2,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                }else{

MPI_Send(&sendBotRow,360,MPI_INT,curIdPs+1,0,MPI_COMM_WORLD);
                    MPI_Recv(&recTopRow,360,MPI_INT,curIdPs-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                    MPI_Send(&sendTopRow,360,MPI_INT,curIdPs-1,0,MPI_COMM_WORLD);

MPI_Recv(&recBotRow,360,MPI_INT,curIdPs+1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                }
            }else{
                if(curIdPs==totNumPs-1){
```

```
                                MPI_Recv(&recTopRow,360,MPI_INT,curIdPs-
1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                                MPI_Send(&sendTopRow,360,MPI_INT,curIdPs-
1,0,MPI_COMM_WORLD);
                        }else{
                                MPI_Recv(&recTopRow,360,MPI_INT,curIdPs-
1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        MPI_Send(&sendBotRow,360,MPI_INT,curIdPs+1,0,MPI_COMM_WORLD);

        MPI_Recv(&recBotRow,360,MPI_INT,curIdPs+1,0,MPI_COMM_WORLD,MPI_STATUS
_IGNORE);
                                MPI_Send(&sendTopRow,360,MPI_INT,curIdPs-
1,0,MPI_COMM_WORLD);
                        }
                }


                int tempArray[rowNum][360];
                for(int t=0;t<rowNum;t++){
                        for(int k=0;k<360;k++){
                                int liveCounter=0;
                                liveCounter+=leftTop(t,k,curIdPs,sendArray,recTopRow);
                                liveCounter+=top(t,k,curIdPs,sendArray,recTopRow);

        liveCounter+=rightTop(t,k,curIdPs,sendArray,recTopRow);
                                liveCounter+=left(t,k,curIdPs,sendArray,recTopRow);
                                liveCounter+=right(t,k,curIdPs,sendArray,recTopRow);

        liveCounter+=botLeft(t,k,curIdPs,sendArray,recBotRow,rowNum,totNumPs);

        liveCounter+=bot(t,k,curIdPs,sendArray,recBotRow,rowNum,totNumPs);

        liveCounter+=botRight(t,k,curIdPs,sendArray,recBotRow,rowNum,totNumPs);
```

```
                                        if(liveCounter<2 && (sendArray[t][k]==1)){
                                                tempArray[t][k]=0;
                                        }else if(liveCounter>3 && (sendArray[t][k]==1)){
                                                tempArray[t][k]=0;
                                        }else if(liveCounter==3 && (sendArray[t][k]==0)){
                                                tempArray[t][k]=1;
                                        }else{
                                                tempArray[t][k]=sendArray[t][k];
                                        }



                                }
                        }
                        for(int t=0;t<rowNum;t++){
                                for(int k=0;k<360;k++){
                                        sendArray[t][k]=tempArray[t][k];
                                }
                        }
                }


                MPI_Send(&sendArray,360*rowNum,MPI_INT,0,0,MPI_COMM_WORLD);


        }
        MPI_Finalize();



        return 0;
}
```