PROGRAMMING PROJECT

Course: CMPE 300 Analysis of Algorithm

Project Title: Parallel Game of Life

Student Name: Ertuğrul Bülbül

Student ID: 2016400219

Submission Date: 22/12/2019

1-Introduction

In this project we are wanted to implement a parallel game by using a specific library which is called MPI. Main purpose of the game is using Message Passing System(MPI) efficiently because it is important to use threads well and ensure a good connection between them is really important in this project. We are wanted to create a special cellular automata which is called Conway's Game of Life, devised by J. H. Conway in 1970. We have a map that contains 1s and 0s which means creature or empty. We simulate system as given turn counter and print the last version of the map. We have some rules for simulation such that if an creature have 4 or more neighbours then it dies,if creature have 1 or 0 neighbour near itself then it dies otherwise it remains alive. For empty part if it has exactly 3 creature in its neighbours then this make a creature in that box. We have main thread and its servants so called workers. Master have a specific job but other servants are doing same job. Master parse the map and send the true part to the servant than servants make some data transfers between each others then make calculations for the maps new situation. Each servant processes want to know which cells are neighbour of its outer most cells. In order to to that one thread must be made some receive operations and some other threds must give data for data waiting thread. Neighbour threads are connecting with each other and send and receive necessary data from the right thread. After data transfers thread can start to calculate its next state through it received necessary information at information transfer step. After calculating its next state it repats all these information transfer and calculate next state steps as turn counter value because turn counter means how many time of calculations must be done. After these calculations finished servant threads send its last state to master thread. After each servant thread sent its final state, master prints last state of the whole map to a file which is taken when the program starting argument.

2-Program Interface and Execution

This program can be run via command line. It needs 1 parameter for first compilation stage and 3 parameters for running step . Compilation stage needs number of threads. And running step need three parameter such that name of input file, name of output file and turn. Input file name is used for taking initial state of the map. In order to start our game we must have an initial state. Output file is not used for an input like thing. It is only used when our program finished its calculation it needs a file name to print its last state. Third parameter which is turn is show us how many time simulate this calculation process.

First command for compilation and create an exacutable:

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

Second command for running our program:

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

[M] : number of threads [T] : Turn counter that shows how many step do we need to simulate game.

3- Input and Output

Program needs three input for execution. First one is input file that contains initial state of the map, second is output file which is only need for name of file and third one is turn counter. Program creates or overwrites one output file for printing last state of the map. Format of the input file and output file are txt and format of the turn counter is integer.

4- Program Structure

First of all I used C++ for coding this project. I take input from an txt file via iostream library than create a big map inside my code as an matrix. Than divide this matrix as small part and send this parts to right thread in order to make calculations. So that means my master only take data from input and parse it for servants. After that my servants start working. They take their matrix part as an linear vector than convert it a matrix shape to make calculations. After that it creates its outer most lines and corners information in order to send data for other servants because in my checkered implementation style corner cells and outer most cells most know other cells information to make calculations. Then it makes send and receive via MPI library and take necessary datas from neighbour cells. In data transfer firstly threads that have odd rank and even rank make their interconnection.

They take their left and right informations. Then threads make other communication but at that time they make their connection according to their row in the big map. In this connection we get top and bottom informations. Behind that we get our corner information. After that we make calculations for each cell. We repeat this data transfer and calculation according to turn counter. After all servant threads send last state of the map to the master. Master takes this maps and refresh big map with its last state. After that it prints last state of the map to an output file that's format is txt. For printing an output file I used fstream library. Also I used several more libraries such as math library for calculating square root, Cstdlib library for using atoi method that enable me to convert taken turn counter to integer and vector library for using vectors.

There are some classes in my project its name is really self explanatory. One of them converts a linear vector to a 2D array. calculateNextTurn() take map and its neighbour cells then calculate the next step of the map. sendInitialStateToThread() method is called in master, it take map and send parts of it to servants. prepareOuterMostLinesToSend() it takes map and prepares outer most lines to send right thread. prepareCornersToSend() takes map then prepare the corners of the matrix.

5-Improvement and Extensions

First of all I implemented project as checkered and in order to ensure connection between processes I made some assumptions by order of pdf that tells us how to do project. The assumption is thread number is always a number which is one more than a perfectly square even number. By using this I implement a logic for my connections. If process number come as an odd number or not a square number then my code would fail because it would probably cause a deadlock in my algorithm.

Then I thought that I could be better in implementing messaging step more modular. I tried to find a similar similarity between source and destination threads which is needed for receive or send but I couldn't find a good path and I thought a better code can be written.

6-Diffuculties Encountered

Before starting implementing my project I tried a code part which is used verry efficiently for all interconnections but I couldn't find a satisfying method but when I started implementing something makes everything easier. Because actually interconnections need some code repats with different numbers because path of the each send and receive are different. Also when I started project I thought that creating a deadlock is very easy case but when implementing it actually it is not easy because if I have done everythin in an order than everything would work clearly

7-Conclusion

I handled problem and submited the project. This Game of Life game is ready for user. The game is working with no errors. It is checkered and periodic. I think I used MPI correctly and finished it.

8-Appendices

```
/* Ertuğrul Bülbül
 * 2016400219
 * Compiling / Working / Checkered / Periodic
*/
#include <iostream>
#include <vector>
#include <mpi.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

/*This method used when all send and recieve operations ended.
```

It takes main matrix as a double pointer and elements that are near that matrix

So it creates a new matrix and calculate new version of the matrix.

Vectors have lines that are near me and other 4 integer value took the elements that is

in the corner of the matrix in the big map*/

```
void calculateNextTurn(int **y,vector<int> &left,vector<int> &right,vector<int> &top,vector<int> &bottom,int tL,int tR,int bL,int bR) {
   int a[left.size()+2][left.size()+2];
   a[0][0]=tL;
   a[0][left.size()+1]=tR;
   a[left.size()+1][0]=bL;
   a[left.size()+1][left.size()+1]=bR;

   for (int i = 0; i < left.size()+2; ++i) {
      for (int j = 0; j < left.size()+2; ++j) {
         if(i==0){
            if(j!=0 && j!=left.size()+1)
               a[i][j]=top[j-1];
         }else if(j==0){
            if(i!=0 && i!=left.size()+1)
               a[i][j]=left[i-1];
         }else if(i==left.size()+1){
            if(j!=0 && j!=left.size()+1)
               a[i][j]=bottom[j-1];
         }else if(j==left.size()+1){
            if(i!=0 && i!=left.size()+1)
               a[i][j]=right[i-1];
         }else{
            a[i][j]=y[i-1][j-1];
         }
      }
   }

   for (int i = 1; i < left.size()+1; ++i) {
      for (int j = 1; j < left.size()+1; ++j) {
         int sum = a[i][j+1]+a[i][j-1]+a[i+1][j]+a[i-1][j]+a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1];
         if(y[i-1][j-1] == 0 && sum == 3){
            y[i-1][j-1] = 1;
         }else if(sum < 2 || sum > 3){
            y[i-1][j-1] = 0;
         }
      }
```

```
    }
}

/*This method takes initial map and send the servants map to right servant as a linear vector
 *It is used in master process*/
void sendInitialStateToThreads(int **map,int matrixPerLine,int elementPerThread,int matrixRowColumn){
    vector<int> toThread(elementPerThread);
    int row = 0, column = 0,i=0,j=0,count = 0,rank = 1;
    while (row < matrixPerLine ){
        if(column == matrixPerLine){
            row++;
            column=0;
        }else{
            while(column<matrixPerLine){
                for (i=matrixRowColumn*(row); i < matrixRowColumn*(row+1); ++i) {
                    for (j= matrixRowColumn*(column); j < matrixRowColumn*(column+1); j++) {
                        toThread[count] = map[i][j];
                        count++;
                    }
                }
                column++;
                count=0;
                MPI_Send(&toThread[0],elementPerThread,MPI_INT,rank,0,MPI_COMM_WORLD);
                rank++;

            }
        }
    }
}

/*This method take an element matrix array and a vector that take information about map state as a linear vector from master.
 * By using this vector my element matrix list is prepared.*/
void linearMapToMatrixConverter(int **elementMatrix,int matrixRowColumn,vector<int> &elementsLinear){
    int temp=0;
    for (int i = 0; i < matrixRowColumn; ++i) {
        for (int j = 0; j < matrixRowColumn; ++j) {
            elementMatrix[i][j] = elementsLinear[temp];
            temp++;
        }
    }
```

```
}

/*Take element matrix and vectors that are going to took information about outer lines of matrix.
 * These vector used when sending data to other threads*/
void prepareOuterMostLinesToSend(int **elementMatrix,int matrixRowColumn,vector<int> &rightMost,vector<int> &leftMost,vector<int> &top,vector<int> &bottom){
   for (int i = 0; i < matrixRowColumn; ++i) {
      top.push_back(elementMatrix[0][i]);
      bottom.push_back(elementMatrix[matrixRowColumn-1][i]);
      leftMost.push_back(elementMatrix[i][0]);
      rightMost.push_back(elementMatrix[i][matrixRowColumn-1]);
   }
}
/* Take element matrix from thread and set its corner values to integer .
 * These integer values used when sending data to other threads.*/
void prepareCornersToSend(int **elementMatrix,int matrixRowColumn,int &topLeftCorner,int &topRightCorner,int &bottomLeftCorner,int &bottomRightCorner){
   topLeftCorner = elementMatrix[0][0];
   topRightCorner = elementMatrix[0][matrixRowColumn-1];
   bottomLeftCorner = elementMatrix[matrixRowColumn-1][0];
   bottomRightCorner = elementMatrix[matrixRowColumn-1][matrixRowColumn-1];
}
int main(int argc,char* argv[]) {
   if(argc != 4){
      cout << "Input file,output file and turn don't given properly."<<endl;
   }
   int myRank,numberOfThread;

   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
   MPI_Comm_size(MPI_COMM_WORLD,&numberOfThread);

   int elementPerThread = 360*360/(numberOfThread-1);
   int matrixRowColumn = sqrt(elementPerThread);
   int matrixPerLine = sqrt(numberOfThread-1);

   //Master process enter this block other threads use else block.
   if(myRank==0){
      int *map[360];
      for (int i = 0; i < 360; ++i) {
```

```
        map[i] = new int[360];
    }
    ifstream input(argv[1]);
    //Take input from file.
    for (int i = 0; i < 360; ++i) {
        for (int j = 0; j < 360; ++j) {
            int data;
            input >> data;
            map[i][j] = data;
        }
    }
    //Master send map of the servants via this method.
    sendInitialStateToThreads(map,matrixPerLine,elementPerThread,matrixRowColumn);
    //Master gets last version of the map from servants.
    vector<int> lastVersionOfMap(elementPerThread);
    for (int k = 1; k < numberOfThread; ++k) {
        MPI_Recv(&lastVersionOfMap[0],elementPerThread,MPI_INT,k,14,MPI_COMM_WORLD,&status);
        int rowOfReceived = (k-1)/matrixPerLine;
        int columOfReceived = (k-1)%matrixPerLine;
        int linearCounter=0;
        for(int l = matrixRowColumn*rowOfReceived; l < matrixRowColumn*(rowOfReceived+1); ++l){
            for(int m = matrixRowColumn*columOfReceived; m < matrixRowColumn*(columOfReceived+1); ++m) {
                map[l][m] = lastVersionOfMap[linearCounter];
                linearCounter++;
            }
        }
    }

    //Print the last version of the map to a file.
    ofstream output(argv[2]);
    for(int n = 0; n < 360; ++n){
        for(int k = 0; k < 360; ++k){
            output << map[n][k] << " ";
        }
        output << "\n";
    }
}else{
    //Servants enter this else block
    //Initial state of the map received as a linear vector.
    vector<int>  elementsLinear(elementPerThread);
```

```
MPI_Recv(&elementsLinear[0],elementPerThread,MPI_INT,0,0,MPI_COMM_WORLD,&status);
int *elementMatrix[matrixRowColumn];
for (int i = 0; i < matrixRowColumn; ++i) {
   elementMatrix[i] = new int[matrixRowColumn];
}
//Convert linear map to matrix.
linearMapToMatrixConverter(elementMatrix,matrixRowColumn,elementsLinear);

//This while loop enable us to make calculations as turn counter value.
int turnCounter = atoi(argv[3]);
while(turnCounter>0){
   turnCounter--;

   //The outer most and corner of the matrix being prepared in order to send other threads.
   vector<int> rightMost, leftMost, top, bottom;
   prepareOuterMostLinesToSend(elementMatrix,matrixRowColumn,rightMost,leftMost,top,bottom);

   int topLeftCorner,topRightCorner,bottomLeftCorner,bottomRightCorner;
   prepareCornersToSend(elementMatrix,matrixRowColumn,topLeftCorner,topRightCorner,bottomLeftCorner,bottomRightCorner);

   //I use this vectors and integers for receiving information from neighbours
   vector<int> myLeft(matrixRowColumn), myRight(matrixRowColumn), myBottom(matrixRowColumn), myTop(matrixRowColumn);
   int myTopLeftCorner,myTopRightCorner,myBottomLeftCorner,myBottomRightCorner;

   //With this if else block all threads make their left-right information exchange.
   if(myRank%2==1){
      //Receive left.
      int sourceLeft = (myRank%matrixPerLine==1) ? myRank+matrixPerLine-1 : sourceLeft=myRank-1;
      MPI_Recv(&myLeft[0],matrixRowColumn,MPI_INT,sourceLeft,1,MPI_COMM_WORLD,&status);
      //Receive right
      int sourceRight = (myRank%matrixPerLine==0) ? myRank-matrixPerLine+1 :sourceRight=myRank+1;
      MPI_Recv(&myRight[0],matrixRowColumn,MPI_INT,sourceRight,2,MPI_COMM_WORLD,&status);
      //Send left
      int destinationLeft=myRank+1;
      MPI_Send(&rightMost[0],matrixRowColumn,MPI_INT,destinationLeft,3,MPI_COMM_WORLD);
      //Send right
      int destinationRight = (myRank%matrixPerLine==1) ? myRank+matrixPerLine-1 : myRank - 1;
      MPI_Send(&leftMost[0],matrixRowColumn,MPI_INT,destinationRight,4,MPI_COMM_WORLD);
   }else{
      //Send left
```

```
    int destinationLeft = (myRank%matrixPerLine==0) ? myRank-matrixPerLine+1 : myRank+1;
    MPI_Send(&rightMost[0],matrixRowColumn,MPI_INT,destinationLeft,1,MPI_COMM_WORLD);
    //Send right
    int destinationRight=myRank-1;
    MPI_Send(&leftMost[0],matrixRowColumn,MPI_INT,destinationRight,2,MPI_COMM_WORLD);
    //Receive left
    int sourceLeft=myRank-1;
    MPI_Recv(&myLeft[0],matrixRowColumn,MPI_INT,sourceLeft,3,MPI_COMM_WORLD,&status);
    //Receive right
    int sourceRight = (myRank%matrixPerLine==0) ? myRank - matrixPerLine + 1 : myRank + 1;
    MPI_Recv(&myRight[0],matrixRowColumn,MPI_INT,sourceRight,4,MPI_COMM_WORLD,&status);
}
/*With this if else block all threads make their top-bottom information exchange
It also makes corner information exchange.
It separates processes as processes that are in the odd row and even row and make information passing and receiving.
*/
if(((myRank-1)/matrixPerLine)%2==0){
    //Receive top information
    int sourceTop = ((myRank-1)/matrixPerLine == 0) ? myRank + matrixPerLine*(matrixPerLine-1) : myRank-matrixPerLine;
    MPI_Recv(&myTop[0],matrixRowColumn,MPI_INT,sourceTop,5,MPI_COMM_WORLD,&status);
    //Receive top-left information
    int sourceTopLeft = (myRank%matrixPerLine==1) ? sourceTop+matrixPerLine-1 : sourceTop-1;
    MPI_Recv(&myTopLeftCorner,1,MPI_INT,sourceTopLeft,10,MPI_COMM_WORLD,&status);
    //Receive top-right information
    int sourceTopRight = (myRank%matrixPerLine==0) ? sourceTop-matrixPerLine+1 : sourceTop+1;
    MPI_Recv(&myTopRightCorner,1,MPI_INT,sourceTopRight,11,MPI_COMM_WORLD,&status);
    //Bottom line information
    int sourceBottom = myRank+matrixPerLine;
    MPI_Recv(&myBottom[0],matrixRowColumn,MPI_INT,sourceBottom,6,MPI_COMM_WORLD,&status);
    //Bottom left
    int sourceBotomLeft = (myRank%matrixPerLine==1) ? sourceBottom + matrixPerLine -1 : sourceBottom -1;
    MPI_Recv(&myBottomLeftCorner,1,MPI_INT,sourceBotomLeft,12,MPI_COMM_WORLD,&status);
    //Bottom right
    int sourceBottomRight = (myRank%matrixPerLine==0) ? sourceBottom - matrixPerLine+1 : sourceBottom + 1;
    MPI_Recv(&myBottomRightCorner,1,MPI_INT,sourceBottomRight,13,MPI_COMM_WORLD,&status);
    //Sending top information
    int destinationTop = myRank + matrixPerLine;
    MPI_Send(&bottom[0],matrixRowColumn,MPI_INT,destinationTop,7,MPI_COMM_WORLD);
    //Sending top-left information
    int destinationTopLeft = (myRank%matrixPerLine==0) ? destinationTop - matrixPerLine+1 :destinationTop +1;
```

```
            MPI_Send(&bottomRightCorner,1,MPI_INT,destinationTopLeft,10,MPI_COMM_WORLD);
            //Sending top-right information
            int destinationTopRight =(myRank%matrixPerLine==1) ? destinationTop+matrixPerLine-1 : destinationTop -1;
            MPI_Send(&bottomLeftCorner,1,MPI_INT,destinationTopRight,11,MPI_COMM_WORLD);
            //Sending bottom information
            int destinationBottom = ((myRank-1)/matrixPerLine == 0) ? myRank + matrixPerLine*(matrixPerLine-1) : myRank - matrixPerLine;
            MPI_Send(&top[0],matrixRowColumn,MPI_INT,destinationBottom,8,MPI_COMM_WORLD);
            //Sending bottomleft
            int destinationBottomLeft = (myRank%matrixPerLine==0) ? destinationBottom - matrixPerLine+1 : destinationBottom+1;
            MPI_Send(&topRightCorner,1,MPI_INT,destinationBottomLeft,12,MPI_COMM_WORLD);
            //Sending bottomright
            int destinationBottomRight = (myRank%matrixPerLine==1) ? destinationBottom +matrixPerLine-1 : destinationBottom-1;
            MPI_Send(&topLeftCorner,1,MPI_INT,destinationBottomRight,13,MPI_COMM_WORLD);
        }else{
            //For sending top of the message waiting threads.
            int destinationTop = ((myRank-1)/matrixPerLine == matrixPerLine-1) ? myRank-matrixPerLine*(matrixPerLine-1) : myRank +
matrixPerLine;
            MPI_Send(&bottom[0],matrixRowColumn,MPI_INT,destinationTop,5,MPI_COMM_WORLD);
            //Send top-left
            int destinationTopLeft = (myRank%matrixPerLine==0) ? destinationTop-matrixPerLine+1 :destinationTop+1;
            MPI_Send(&bottomRightCorner,1,MPI_INT,destinationTopLeft,10,MPI_COMM_WORLD);
            //Send top-right
            int destinationTopRight = (myRank%matrixPerLine==1) ? destinationTop+matrixPerLine-1 : destinationTop-1;
            MPI_Send(&bottomLeftCorner,1,MPI_INT,destinationTopRight,11,MPI_COMM_WORLD);
            //Bottom information
            int destinationBottom = myRank-matrixPerLine;
            MPI_Send(&top[0],matrixRowColumn,MPI_INT,destinationBottom,6,MPI_COMM_WORLD);
            //Send bottom left
            int destinationBottomLeft = (myRank%matrixPerLine==0) ? destinationBottom-matrixPerLine +1 : destinationBottom +1;
            MPI_Send(&topRightCorner,1,MPI_INT,destinationBottomLeft,12,MPI_COMM_WORLD);
            //Send bottom right
            int destinationBottomRight = (myRank%matrixPerLine==1) ? destinationBottom + matrixPerLine -1 : destinationBottom -1;
            MPI_Send(&topLeftCorner,1,MPI_INT,destinationBottomRight,13,MPI_COMM_WORLD);
            //Recieve top
            int sourceTop = myRank - matrixPerLine;
            MPI_Recv(&myTop[0],matrixRowColumn,MPI_INT,sourceTop,7,MPI_COMM_WORLD,&status);
            //Receive top-left
            int sourceTopLeft = (myRank%matrixPerLine==1) ? sourceTop+matrixPerLine-1 : sourceTop-1;
            MPI_Recv(&myTopLeftCorner,1,MPI_INT,sourceTopLeft,10,MPI_COMM_WORLD,&status);
            //Receive top-right
            int sourceTopRight = (myRank%matrixPerLine==0) ? sourceTop-matrixPerLine +1 : sourceTop +1;
```

```
            MPI_Recv(&myTopRightCorner,1,MPI_INT,sourceTopRight,11,MPI_COMM_WORLD,&status);

            //Receive bottom

            int sourceBottom = ((myRank-1)/matrixPerLine == matrixPerLine-1) ? myRank-matrixPerLine*(matrixPerLine-1) : myRank +
matrixPerLine;

            MPI_Recv(&myBottom[0],matrixRowColumn,MPI_INT,sourceBottom,8,MPI_COMM_WORLD,&status);

            //Receive bottom-left

            int sourceBottomLeft = (myRank%matrixPerLine==1) ? sourceBottom+matrixPerLine-1 : sourceBottom-1;

            MPI_Recv(&myBottomLeftCorner,1,MPI_INT,sourceBottomLeft,12,MPI_COMM_WORLD,&status);

            //Receive bottom-right

            int sourceBottomRight = (myRank%matrixPerLine==0) ? sourceBottom-matrixPerLine +1 : sourceBottom +1;

            MPI_Recv(&myBottomRightCorner,1,MPI_INT,sourceBottomRight,13,MPI_COMM_WORLD,&status);

          }

calculateNextTurn(elementMatrix,myLeft,myRight,myTop,myBottom,myTopLeftCorner,myTopRightCorner,myBottomLeftCorner,myBotto
mRightCorner);

        }

        //If process is not master process it sends its map information to master.

        if(myRank!=0){

          vector<int> toMaster(elementPerThread);

          int variable = 0;

          for (int i = 0; i < matrixRowColumn; ++i) {

            for (int j = 0; j < matrixRowColumn; ++j) {

              toMaster[variable] = elementMatrix[i][j];

              variable++;

            }

          }

          MPI_Send(&toMaster[0],elementPerThread,MPI_INT,0,14,MPI_COMM_WORLD);

        }

      }

      MPI_Finalize();

      return 0;

}
```