CmpE 300
Analysis of Algorithms
Fall 2019
Programming Project
Conway's Game of Life

20.12.2019

Emilcan ARICAN - 2016400231

1 Introduction

1.1 Cellular Automata

Cellular automata is a discrete model used in computer science, mathematics, physics etc. Enables us to observe emergence.

It consists a grid of cells that have finite number of states. For each cell certain cells are called as neighbors. Next state of a cell is determined according to its own state and state of its neighbors. When simulation is at the time t, the state at the t+1 can be derived. Simulation starts from t = 0, and ends at the time t = T.

1.2 Conway's Game of Life

Subject of this project is Conway's Game of Life that is designed in 1970 by J. H. Conway. Characteristics and rules are like the stated below:

- Each cell has 8 neighbors that surrounds them.

$$\begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & N & N & N & .. \\ .. & N & A & N & .. \\ .. & N & N & N & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix}$$

> *N's are neighbors of the A.*

- Each cell has two cells.
  - 0 $\rightarrow$ cell contains no life
  - 1 $\rightarrow$ cell contains creature

1.2.1 Rules

- **Loneliness:** If creature has less than 2 creature neighbors, it dies due to loneliness.

$$\begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & 0 & 0 & 1 & .. \\ .. & 0 & 1 & 0 & .. \\ .. & 0 & 0 & 0 & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix} \rightarrow \begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & ? & ? & ? & .. \\ .. & ? & 0 & ? & .. \\ .. & ? & ? & ? & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix}$$

- **Overpopulation:** If creature has more than 3 creature neighbors, it dies due to overpopulation.

$$\begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & 0 & 0 & 1 & .. \\ .. & 0 & 1 & 0 & .. \\ .. & 1 & 0 & 1 & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix} \rightarrow \begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & ? & ? & ? & .. \\ .. & ? & 0 & ? & .. \\ .. & ? & ? & ? & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix}$$

- **Reproduction:** If an empty cell has 3 creature neighbors, they reproduce, so this cell contains creature.

$$\begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & 0 & 0 & 1 & .. \\ .. & 0 & 0 & 0 & .. \\ .. & 1 & 0 & 1 & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix} \rightarrow \begin{bmatrix} & \vdots & \vdots & \vdots & \\ .. & ? & ? & ? & .. \\ .. & ? & 1 & ? & .. \\ .. & ? & ? & ? & .. \\ & \vdots & \vdots & \vdots & \end{bmatrix}$$

- **Others:** In the other cases, cells remains as they are. If it contains a creature, it will contain at next state; if it is empty, it will be empty at next state.

> *Since, next state of the neighbors, depends on their neighbors they are demonstrated with '?' symbol.*

1.2.2 Boundaries

A toroidal map is used in this project. Toroidal map is a map that edges connected (left-right; up-down). In order to conceptualize it in the mind, fist bend map into cylindrical shape. Then, bend that cylindrical shape into donut like shape. So some neighbors of certain cells are at the other side of the map.

2 Program Interface

2.1 Dependencies

In order to setup the *mpi4py* these command is used.

```
sudo apt install libopenmpi-dev
sudo pip3 install mpi4py
```

numpy is used for array manipulations. Therefore it should be installed too.

```
sudo pip3 install numpy
```

2.2 Execution

User can run the program with the command below.

```
mpirun --oversubscribe -np <process-amount> python3 main.py <input-file> <output-file> <T>
```

- **<process-amount>** : Number of processes will be used while executing.
- **<input-file>** : File that contains input.
- **<output-file>** : File that output will be written into
- **<T>** : Number of iteration that will be performed; in other words, desired state.

3 Program Execution

Program takes an input file that contains a binary matrix. It defines the 0th state. Then, according to rules of the Conway's Game of Life calculates the next states of it by using defined amount of processes, until the desired state. When it reaches the desired state prints it into the output file.

4 Input and Output

Input and output files contains matrices that represents the cells contains creatures and the empty ones. File format is stated below:

- File contains values of 0 and 1. 0 is for empty; 1 is for alive.
- Each element in the rows separated with spaces.
- Each row separated from the other rows with ('\n') newline character.

5 Program Structure

5.1 Read Input

First of all manager reads input into a numpy array.

5.2 Split and Send the Chunks

Then manager splits the map into disjoint square chunks and sends them to workers.

In order to take the sub arrays properly, there are 4 utility functions (*widthBegin, widthEnd, heightBegin, heightEnd*).These defines where should it start and where should it end according to rank of the process.

5.3 Worker Communications

In order to determine the rank of the neighbor processes there are 8 utility function namely: *upChunk, downChunk, leftChunk, rightChunk, upLeftChunk, upRightChunk, downLeftChunk, downRightChunk*. These allows workers to know neighbor processes ranks according to their own rank.

In order to identify type of the process *isWhite* and *isEven* functions are used.

In order to send and receive messages *sendHorizontalAndVertical, recvHorizontalAndVertical, sendDiagonal, recvDiagonal* functions are used.

> *Chess board analogy is used. Therefore some processes are called White and the others Black.*

5.3.1 Horizontal and Vertical

White processes sends their border cells vertically and horizontally (up border to up neighbor and so on). Black processes receive these border cells.

Then Blacks send and Whites receive.

5.3.2 Diagonal

First even ranked processes sends corner values to neighbor at the same corner. The processes with odd rank receive the values.

Then odd processes send and even processes receive.

5.4 Calculating Next State

Worker concatenates the received arrays with the chunk it has. So, there are no cells with missing neighbors. While iterating over the cells in the chunk for every cell first neighbor number is calculated via *neighborNum* function, then next state is determined with *nextState* function that takes neighbor number and current state of the cell as an input. When chunk of the next state is determined, iteration continues until the desired state.

5.5 Collecting Back

When desired state is calculated. Workers sends their chunks to the manager. Manager collects the chunks and concatenates them into one matrix. So final map is formed.

5.6 Write Output

Writes final map into output file. With space separated characters and rows divided by newline character.

6 Conclusion

Program that is an implementation of Conway's Game of Life, is working properly and uses benefits of the multiprocessing.

7 Appendix

```
# Emilcan Arıcan
# 2016400231
# Compiling
# Working

from mpi4py import MPI
import numpy as np
import math
import sys

comm = MPI.COMM_WORLD

T           = int(sys.argv[3])
rank        = comm.Get_rank()
max_rank    = comm.size - 1

#number of chunks along one edge of the map
period      = int(math.sqrt(max_rank))

#length of the one side of a chunk
```

```
chunk_edge = int(360//period)

#functions that calculate the ranks of the neighbor processes
def upChunk(r):
    if r <= period:
        return r - period + max_rank
    else:
        return r - period

def downChunk(r):
    if (r > (max_rank - period)):
        return r + period - max_rank
    else:
        return r + period

def leftChunk(r):
    if r % period == 1:
        return r - 1 + period
    else:
        return r - 1

def rightChunk(r):
    if r % period == 0:
        return r + 1 - period
    else:
        return r + 1

def upLeftChunk(r):
    return leftChunk(upChunk(r))

def upRightChunk(r):
    return rightChunk(upChunk(r))

def downLeftChunk(r):
    return leftChunk(downChunk(r))

def downRightChunk(r):
    return rightChunk(downChunk(r))

# functions that calculate where chunks should be started and be ended
at
def widthBegin(r):
    return chunk_edge*((r-1)%period)

def widthEnd(r):
    return chunk_edge*(((r-1)%period)+1)

def heightBegin(r):
    return chunk_edge*(((r-1)//period))

def heightEnd(r):
    return chunk_edge*(((r-1)//period)+1)

#function that checks if it is a white ranked process
def isWhite():
    return (((rank-1)%period)+((rank-1)//period))%2 == 1

#function that checks if it is a even ranked process
def isEven():
    return rank%2 == 0

#function that determines the next state of a location
#depending on current state and the number of neighbors
def nextState(currentState, neighborNum):
    if currentState == 1:
        if neighborNum < 2:
            return 0
        elif neighborNum > 3:
            return 0
        else:
            return 1
    elif currentState == 0:
        if neighborNum == 3:
            return 1
        else:
            return 0

#calculates the number of the neighbors
def neighborNum(_map, i, j):
    return (_map[i-1][j-1]
            + _map[i-1][j]
            + _map[i][j-1]
            + _map[i+1][j]
            + _map[i][j+1]
            + _map[i+1][j+1]
            + _map[i-1][j+1]
            + _map[i+1][j-1])

def main():
    #manager
    if rank == 0:

        #input-outputfiles
        inputfile   = sys.argv[1]
        outputfile  = sys.argv[2]

        #read input from file to numpy array
        map = np.loadtxt(inputfile, dtype=int)

        #split map into chunks; send chunks to workers
        for r in range(1,max_rank+1):

            data = map[heightBegin(r):heightEnd(r) ,
widthBegin(r):widthEnd(r)].copy()

            comm.Send(data, dest=r, tag=0)

        #variable that holds concatenated chunks
        chunks = np.empty(shape=[chunk_edge,chunk_edge*period],
dtype=int)
```

```
        #receive chunks from workers and concatenate them
        for i in range(0,period):

            #temporary variable that holds concatenated chunks
            temp = np.empty(shape=[chunk_edge,chunk_edge], dtype=int)

            for j in range(0,period):

                #receive data from worker
                data = np.empty(shape=[chunk_edge,chunk_edge],
dtype=int)
                comm.Recv(data, source=(period*i+j+1))

                #concatenate
                if j == 0:
                    temp = data
                else:
                    temp = np.concatenate((temp,data),axis=1)

            #concatenate
            if i == 0:
                chunks = temp
            else:
                chunks = np.concatenate((chunks,temp))

        #save into file
        np.savetxt(outputfile, chunks, delimiter=" ", fmt='%i',
newline=" \n")

    #worker
    if rank > 0:

        #receive data from the manager
        data = np.empty(shape=[chunk_edge,chunk_edge], dtype=int)
        comm.Recv(data, source=0, tag=0)

        #send horizontal and vertical function
        def sendHorizontalAndVertical(tag1, tag2, tag3, tag4):
            comm.Send(upRow_s,      dest=upChunk(rank), tag=tag1)
            comm.Send(downRow_s,    dest=downChunk(rank), tag=tag2)
            comm.Send(leftCol_s,    dest=leftChunk(rank), tag=tag3)
            comm.Send(rightCol_s,   dest=rightChunk(rank), tag=tag4)

        #receive horizontal and vertical function
        def recvHorizontalAndVertical(tag1, tag2, tag3, tag4):
            comm.Recv(downRow_r,    source=downChunk(rank), tag=tag1)
            comm.Recv(upRow_r,      source=upChunk(rank), tag=tag2)
            comm.Recv(rightCol_r,   source=rightChunk(rank), tag=tag3)
            comm.Recv(leftCol_r,    source=leftChunk(rank), tag=tag4)

        #send diagonal function
        def sendDiagonal(tag1, tag2, tag3, tag4):
            comm.Send(upRight_s,    dest=upRightChunk(rank), tag=tag1)
            comm.Send(upLeft_s,     dest=upLeftChunk(rank), tag=tag2)
            comm.Send(downRight_s,  dest=downRightChunk(rank),
tag=tag3)
            comm.Send(downLeft_s,   dest=downLeftChunk(rank),
tag=tag4)

        #receive diagonal function
        def recvDiagonal(tag1, tag2, tag3, tag4):
            comm.Recv(downLeft_r,   source=downLeftChunk(rank),
tag=tag1)
            comm.Recv(downRight_r,  source=downRightChunk(rank),
tag=tag2)
            comm.Recv(upLeft_r,     source=upLeftChunk(rank),
tag=tag3)
            comm.Recv(upRight_r,    source=upRightChunk(rank),
tag=tag4)

        for _ in range(0,T):

            #variables that will be sent
            upRow_s     = data[0,:].copy()
            downRow_s   = data[-1,:].copy()
            leftCol_s   = data[:,0].copy()
            rightCol_s  = data[:,-1].copy()
            upLeft_s    = data[0,0].copy()
            upRight_s   = data[0,-1].copy()
            downLeft_s  = data[-1,0].copy()
            downRight_s = data[-1,-1].copy()

            #variables that will be received
            upRow_r     = np.empty(chunk_edge, dtype=int)
            downRow_r   = np.empty(chunk_edge, dtype=int)
            leftCol_r   = np.empty(shape=[1,chunk_edge], dtype=int)
            rightCol_r  = np.empty(shape=[1,chunk_edge], dtype=int)
            upLeft_r    = np.empty(1, dtype=int)
            upRight_r   = np.empty(1, dtype=int)
            downLeft_r  = np.empty(1, dtype=int)
            downRight_r = np.empty(1, dtype=int)

            #white tiles (like a chess board)
            if isWhite():
                sendHorizontalAndVertical(1,2,3,4)
                recvHorizontalAndVertical(5,6,7,8)

            #black tiles (like a chess board)
            else:
                recvHorizontalAndVertical(1,2,3,4)
                sendHorizontalAndVertical(5,6,7,8)

            #even ranked tiles
            if isEven():
                sendDiagonal(9,10,11,12)
                recvDiagonal(13,14,15,16)

            #odd ranked tiles
            else:
                recvDiagonal(9,10,11,12)
                sendDiagonal(13,14,15,16)

            #concatenate parts that collected from neighbors with data
            top_row = np.concatenate((upLeft_r,upRow_r,upRight_r))
            mid_rows = 
np.concatenate((np.transpose(leftCol_r),data,np.transpose(rightCol_r))
, axis=1)
            bot_row = 
np.concatenate((downLeft_r,downRow_r,downRight_r))
            framed_data = 
np.concatenate((np.transpose(top_row[:,None]),mid_rows,np.transpose(bo
t_row[:,None])))

            #create next_data that next state will be calculated in
            next_data = np.empty(shape=[chunk_edge,chunk_edge],
dtype=int)

            for i in range(1,chunk_edge+1):
                for j in range(1,chunk_edge+1):

                    #number of neighbors
                    neighbor_num = neighborNum(framed_data, i, j)

                    #fill the next_data array
                    next_data[i-1][j-1] = nextState(framed_data[i][j],
neighbor_num)

            #assign next_data to data for the next iteration
            data = next_data.copy()

        #send the final result to manager
        comm.Send(data, dest=0, tag=17)

main()
```