Parallel Game of Life

CMPE 300: Analysis of Algorithms

Ramiz Dündar

2016400012

18 December 2019

MPI Programming Assignment

Introduction

A cellular automaton is a discrete computational model used in the study of complex systems. It consists of an N-dimensional orthogonal grid called the “map”, and rules of evolution. These alone define our complex environment.

We then assign some initial values to the locations on the map, and make our system evolve by our set of rules via simulation:

1. The map’s state at time t = 0 is as initialized.

2. To calculate the map’s state at time t = 1, we look at the state at time t = 0. For each cell on the map at time t = 1, we look at the same cell and its neighbors at time t = 0.

3. We keep on simulating like this until time t = T .

Game of life is such a cellular automation problem. The state transitions of the cells are given below.

1. Loneliness kills: A creature dies (i.e. the cell becomes empty) if it has less than 2 neighboring creatures.

2. Overpopulation also kills: A creature dies (and becomes empty) if it has more than 3 neighboring creatures. See the following example. Note that the creature would not die if it had one less neighboring creature.

3. Reproduction: A new life appears on an empty cell if it has exactly 3 neighboring creatures. See the following example. Note that the creature would not be born if the cell had one more or one less neighboring creature.

4. In any other condition, the creatures remain alive, and the empty spaces remain empty.

To solve this problem processes number of processes are created each calculates some part of the map. These processes partitioned into 4 parts and then MPI protocols are done. Thus program runs “num_processes” different process at the same time without deadlocks. Note that this variable and how to set it will be explained later.

Program Interface

The input is given through command line interface(CLI). It must be executed through python 3 and paths must be relative to the current directory or absolute paths. Note that numpy library and mpi4py library also must be installed on python 3. To ensure this pip3 could be used. An example run command should be like this:

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

Note that here python should be python 3. M is number is number of processes which is indicated by “num_processes” variable in the code. T is number of iterations(“T” variable in the code)

To terminate the code user can press Ctrl + C once. To force it press again. On Mac use Cmd instead of Ctrl.

Program Execution

Input should be given in .txt format. It must be a file such that every one of 360 row has 360 integer elements which can only be 0 or 1 separated by space. Output is a .txt file with same properties as input file. Program calculates state of the matrix given in the input file after T iterations according to game of life. This is done by M processes concurrently. User can specify input, T, M and output path through CLI.

Input and Output

Input should be given in .txt format. It must be a file such that every one of 360 row has 360 integer elements which can only be 0 or 1 separated by space. Output is a .txt file with same properties as input file.

Program Structure

Libraries Used:

mpi4py: This is mpi api for python. Used for parallelization of the program.

numpy: mpi requires c continuous data structures to be send. Unfortunately default python lists doesn't have this property. Numpy arrays are both memory efficient and C continuous. C continuous here means consecutive elements in the array are saved to consecutive locations.

sys: Standard python library. Used for taking command line arguments.

Useful Variables:

rank: This is for understanding which process we are in the code. Process number.

num_processes: An integer for seeing how many processes are there.

C: The map will be split into C^2 square arrays, each with S/C rows and columns. Used C differently from the project description(C as c in the description).

N: Number of rows and columns in the map.

T: Number of iterations game will be played.

size: Row and column size of the any part map is divided.

grid: Square matrix which has size+2 rows. Additional 2 rows are for ease in implementations when messages received from other processes. Basically map for each process. Note that grid for main is whole map and different from other grids size-wise.

recieved_from: A dictionary for receiving messages. Defined because it makes receiving implementation much easier.

Target and Tag variables: Series of variables for avoiding unnecessary operations and increasing readability.

Functions:

update_message: Updating current message to be send from current grid.

grid_update: Updating grid with messages received from other processes.

total_neighbour: A helper function for returning total scores around a cell.

game_loop: After updating grid makes one iteration of the game according to rules.

passing_{1, 2, 3, 4}: To further understand these functions please look at logic section. Basically these functions manages send and receive operations.

recv_from_parent: Function receives parts of full map from rank zero process and places it into grids.

send_to_child: For sending parts of full map to non-zero rank processes.

send_to_main: Senders to main after all iterations done.

recv_from_child: Receives all small grids from non-zero rank processes.

Logic:

In this subsection general logic behind the code is discussed. First input is taken with rank zero process(parent). After that input is distributed to child processes(non-zero rank processes). In the child processes after getting necessary information from neighboring processes game loop executed. This is repeated T time and then small grids returned to parent. After that output file is created and last state of the map is saved there.

Main problem is preventing deadlock when processes are communicating. To ensure this children divided into 4 parts according to their row and columns. Every even row and column is one group, every odd column and even row is one group and etc. For example when C is 4 map will look like this:

1 2 1 2

3 4 3 4

1 2 1 2

3 4 3 4

This ensures no children group has neighboring child from same group. Thus all processes in the same group send and receive message same way. After this problem reduced to just managing transactions between these four groups. Order of cheese transactions doesn't matter as long as if we ensure when one of the groups send there will ben one group receive that. To do this first one group is send and receives in any order. Then second group will do this but it first does first groups request. And after third group first completes first two group's transactions etc.

Example

The ‘*’ represent an alive cell and the ‘.’ represent a dead cell.

Input and after one iteration output is given:

```
Input : ..........
        ...**.....
        ....*.....
        ..........
        ..........
```

```
Output: ..........
        ...**.....
        ...**.....
        ..........
        ..........
        ..........
```

Improvements and Extensions

Moving to C++ could be a huge improvement.

Using booleans instead of 8 bint integers reduce memory consumption by a great margin.

Using less functions call albeit reducing readability could increase performance

Conclusions

Parallel programming is a mind-expanding area of expertise in computer science. It requires great planning and utmost attention due to hardships of duplicating previous errors. Most of my time spent on checking if there are any deadlock possibility in the code. Although increasing parallelization may seem good idea at first one must not forget that with limited cores increasing it after some threshold will only slow the program.

Appendix

Code:

```
# Ramiz Dündar 2016400012
# Compiling
# Working
# Note that method names and variable names are self explanatory
# If you need further understanding please take a look the project
# report where every variable and function is explained in detail
from mpi4py import MPI
import numpy as np
import sys
import pdb
np.set_printoptions(edgeitems=10)
# mpirun -np 17 python game.py input.txt output.txt 5

# necessary variables for all processes
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
num_processes = comm.Get_size()
C = np.sqrt(num_processes - 1)
N = 360
T = int(sys.argv[3])
# size of small grid inside one process (without adding messages)
size = int(N / C)
grid = np.zeros((size+2, size+2), dtype=np.int8)
recieved_from = {
        'up' : np.empty(size, dtype=np.int8),
        'down' : np.empty(size, dtype=np.int8),
        'left' : np.empty(size, dtype=np.int8),
        'right' : np.empty(size, dtype=np.int8),
        'leftup' : np.empty((), dtype=np.int8),
        'rightup' : np.empty((), dtype=np.int8),
        'leftdown' : np.empty((), dtype=np.int8),
        'rightdown' : np.empty((), dtype=np.int8)
}


# variables for targets and tags
rank -= 1

up_target = (rank - C) % C**2 + 1
up_tag = 1
down_target = (rank + C) % C**2 + 1
down_tag = 2
left_target = (rank//C)*C + (rank-1)%C + 1
left_tag = 3
right_target = (rank//C)*C + (rank+1)%C + 1
right_tag = 4
```

```
leftup_target = ((rank//C-1)%C)*C + (rank-1)%C + 1
leftup_tag = 5
rightup_target = ((rank//C-1)%C)*C + (rank+1)%C + 1
rightup_tag = 6
leftdown_target = ((rank//C+1)%C)*C + (rank-1)%C + 1
leftdown_tag = 7
rightdown_target = ((rank//C+1)%C)*C + (rank+1)%C + 1
rightdown_tag = 8

rank += 1


def update_messages(rank,grid):

    up_message = np.array(grid[1, 1:1+size], dtype=np.int8)
    down_message = np.array(grid[size, 1:1+size], dtype=np.int8)
    left_message = np.array(grid[1:1+size, 1], dtype=np.int8)
    right_message = np.array(grid[1:1+size, size], dtype=np.int8)

    leftup_message = np.array(grid[1, 1], dtype=np.int8)
    rightup_message = np.array(grid[1, size], dtype=np.int8)
    leftdown_message = np.array(grid[size, 1], dtype=np.int8)
    rightdown_message = np.array(grid[size, size], dtype=np.int8)

    messages = {
        'up' : up_message,
        'down' : down_message,
        'left' : left_message,
        'right' : right_message,
        'leftup' : leftup_message,
        'rightup' : rightup_message,
        'leftdown' : leftdown_message,
        'rightdown' : rightdown_message
    }

    return messages


def grid_update(recieved_from, grid):

    grid[0, 1:1+size] = recieved_from['up']
    grid[-1, 1:1+size] = recieved_from['down']
    grid[1:1+size, 0] = recieved_from['left']
    grid[1:1+size, -1] = recieved_from['right']

    grid[0, 0] = recieved_from['leftup']
    grid[0, -1] = recieved_from['rightup']
    grid[-1, 0] = recieved_from['leftdown']
    grid[-1, -1] = recieved_from['rightdown']
```

```
def total_neigbour(i, j, grid):

    tot = (
        grid[i-1][j-1]
        + grid[i-1][j]
        + grid[i-1][j+1]
        + grid[i][j-1]
        + grid[i][j+1]
        + grid[i+1][j-1]
        + grid[i+1][j]
        + grid[i+1][j+1]
    )

    return tot


def game_loop(grid):

    temp_grid = np.array(grid[1:1+size, 1:1+size])

    for i in range(size):
        for j in range(size):

            total = total_neigbour(i+1, j+1, grid)

            if total < 2:
                temp_grid[i][j] = 0
            if total > 3:
                temp_grid[i][j] = 0
            if total == 3:
                temp_grid[i][j] = 1


    grid[1:1+size, 1:1+size] = temp_grid


def passing_1(rank, grid):

    messages = update_messages(rank, grid)

    comm.Send(messages['up'], dest=up_target, tag=up_tag)
    comm.Send(messages['down'], dest=down_target, tag=down_tag)
    comm.Send(messages['left'], dest=left_target, tag=left_tag)
    comm.Send(messages['right'], dest=right_target, tag=right_tag)
    comm.Send(messages['leftup'], dest=leftup_target, tag=leftup_tag)
    comm.Send(messages['rightup'], dest=rightup_target, tag=rightup_tag)
    comm.Send(messages['leftdown'], dest=leftdown_target, tag=leftdown_tag)
    comm.Send(messages['rightdown'], dest=rightdown_target, tag=rightdown_tag)
```

```
    comm.Recv(recieved_from['up'], source=up_target, tag=down_tag)
    comm.Recv(recieved_from['down'], source=down_target, tag=up_tag)
    comm.Recv(recieved_from['left'], source=left_target, tag=right_tag)
    comm.Recv(recieved_from['right'], source=right_target, tag=left_tag)
    comm.Recv(recieved_from['leftup'], source=leftup_target, tag=rightdown_tag)
    comm.Recv(recieved_from['rightup'], source=rightup_target, tag=leftdown_tag)
    comm.Recv(recieved_from['leftdown'], source=leftdown_target, tag=rightup_tag)
    comm.Recv(recieved_from['rightdown'], source=rightdown_target, tag=leftup_tag)

    grid_update(recieved_from, grid)

    game_loop(grid)


def passing_2(rank, grid):

    messages = update_messages(rank, grid)

    comm.Recv(recieved_from['right'], source=right_target, tag=left_tag)
    comm.Recv(recieved_from['left'], source=left_target, tag=right_tag)

    comm.Send(messages['right'], dest=right_target, tag=right_tag)
    comm.Send(messages['left'], dest=left_target, tag=left_tag)

    comm.Send(messages['up'], dest=up_target, tag=up_tag)
    comm.Send(messages['down'], dest=down_target, tag=down_tag)
    comm.Send(messages['leftup'], dest=leftup_target, tag=leftup_tag)
    comm.Send(messages['rightup'], dest=rightup_target, tag=rightup_tag)
    comm.Send(messages['leftdown'], dest=leftdown_target, tag=leftdown_tag)
    comm.Send(messages['rightdown'], dest=rightdown_target, tag=rightdown_tag)

    comm.Recv(recieved_from['up'], source=up_target, tag=down_tag)
    comm.Recv(recieved_from['down'], source=down_target, tag=up_tag)
    comm.Recv(recieved_from['leftup'], source=leftup_target, tag=rightdown_tag)
    comm.Recv(recieved_from['rightup'], source=rightup_target, tag=leftdown_tag)
    comm.Recv(recieved_from['leftdown'], source=leftdown_target, tag=rightup_tag)
    comm.Recv(recieved_from['rightdown'], source=rightdown_target, tag=leftup_tag)

    grid_update(recieved_from, grid)

    game_loop(grid)


def passing_3(rank, grid):

    messages = update_messages(rank, grid)

    comm.Recv(recieved_from['down'], source=down_target, tag=up_tag)
    comm.Recv(recieved_from['up'], source=up_target, tag=down_tag)
```

```
    comm.Recv(recieved_from['rightdown'], source=rightdown_target, tag=leftup_tag)
    comm.Recv(recieved_from['leftdown'], source=leftdown_target, tag=rightup_tag)
    comm.Recv(recieved_from['rightup'], source=rightup_target, tag=leftdown_tag)
    comm.Recv(recieved_from['leftup'], source=leftup_target, tag=rightdown_tag)

    comm.Send(messages['down'], dest=down_target, tag=down_tag)
    comm.Send(messages['up'], dest=up_target, tag=up_tag)

    comm.Send(messages['rightdown'], dest=rightdown_target, tag=rightdown_tag)
    comm.Send(messages['leftdown'], dest=leftdown_target, tag=leftdown_tag)
    comm.Send(messages['rightup'], dest=rightup_target, tag=rightup_tag)
    comm.Send(messages['leftup'], dest=leftup_target, tag=leftup_tag)

    comm.Send(messages['left'], dest=left_target, tag=left_tag)
    comm.Send(messages['right'], dest=right_target, tag=right_tag)

    comm.Recv(recieved_from['left'], source=left_target, tag=right_tag)
    comm.Recv(recieved_from['right'], source=right_target, tag=left_tag)

    grid_update(recieved_from, grid)

    game_loop(grid)


def passing_4(rank, grid):

    messages = update_messages(rank, grid)

    comm.Recv(recieved_from['rightdown'], source=rightdown_target, tag=leftup_tag)
    comm.Recv(recieved_from['leftdown'], source=leftdown_target, tag=rightup_tag)
    comm.Recv(recieved_from['rightup'], source=rightup_target, tag=leftdown_tag)
    comm.Recv(recieved_from['leftup'], source=leftup_target, tag=rightdown_tag)

    comm.Recv(recieved_from['down'], source=down_target, tag=up_tag)
    comm.Recv(recieved_from['up'], source=up_target, tag=down_tag)

    comm.Recv(recieved_from['right'], source=right_target, tag=left_tag)
    comm.Recv(recieved_from['left'], source=left_target, tag=right_tag)

    comm.Send(messages['rightdown'], dest=rightdown_target, tag=rightdown_tag)
    comm.Send(messages['leftdown'], dest=leftdown_target, tag=leftdown_tag)
    comm.Send(messages['rightup'], dest=rightup_target, tag=rightup_tag)
    comm.Send(messages['leftup'], dest=leftup_target, tag=leftup_tag)

    comm.Send(messages['down'], dest=down_target, tag=down_tag)
    comm.Send(messages['up'], dest=up_target, tag=up_tag)

    comm.Send(messages['right'], dest=right_target, tag=right_tag)
    comm.Send(messages['left'], dest=left_target, tag=left_tag)
```

```
    grid_update(recieved_from, grid)

    game_loop(grid)


def recv_from_parent(grid):
    message = np.empty((size, size), dtype=np.int8)
    comm.Recv(message, source=0)
    grid[1:1+size, 1:1+size] = message


def send_to_child(child_rank, grid):
    row = int((child_rank-1) // C) * size
    col = int((child_rank-1) % C) * size
    message = np.array(grid[row:row+size, col:col+size], dtype=np.int8)
    comm.Send(message, dest=child_rank)

def send_to_main(grid):
    message = np.array(grid[1:1+size, 1:1+size])
    comm.Send(message, dest=0)

def recv_from_child(child_rank, grid):
    message = np.empty((size, size), dtype=np.int8)
    comm.Recv(message, source=child_rank)
    row = int((child_rank-1) // C) * size
    col = int((child_rank-1) % C) * size
    grid[row:row+size, col:col+size] = message

if rank != 0:
    recv_from_parent(grid)
    row = int((rank-1) // C)
    col = int((rank-1) % C)

if rank == 0:

    if len(sys.argv) != 4:
        print('argument count error')

    grid = np.zeros((N, N), dtype=np.int8)

    try:
        inp = open(sys.argv[1])
        for i, line in enumerate(inp):
            row = [int(x) for x in line.split()]
            grid[i] = np.array(row)
    except FileNotFoundError:
        print('File not accessible')

    # check c is integer and even
    if not C.is_integer():
```

```
        print('C is not an integer')
        print(f'number of processes is: {num_processes}')
        raise ValueError
    elif C % 2 == 1:
        print('C is not even')
        print(f'number of processes is: {num_processes}')
        raise ValueError
    else:
        C = int(C)

    for child_rank in range(1, num_processes):
        send_to_child(child_rank, grid)

    for child_rank in range(1, num_processes):
        recv_from_child(child_rank, grid)

    np.savetxt(sys.argv[2], grid, fmt='%d', encoding='utf-8', newline=' \n')

elif row % 2 == 0 or col % 2 == 0:
    for _ in range(T):
        passing_1(rank, grid)

elif row % 2 == 0 or col % 2 == 1:
    for _ in range(T):
        passing_2(rank, grid)

elif row % 2 == 1 or col % 2 == 0:
    for _ in range(T):
        passing_3(rank, grid)

elif row % 2 == 1 or col % 2 == 1:
    for _ in range(T):
        passing_4(rank, grid)

else:
    print('code should not reach here')
    raise RuntimeError

if rank != 0:
    send_to_main(grid)
```