Boğaziçi University
CMPE 300 - Analysis of Algorithms
Fall 2019
Parallel Game of Life
Programming Project
Tunga Güngör

Baran Deniz Korkmaz - 2015400183

December 22, 2019

Contents

1 Introduction

1.1 Definition

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.
The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The project aims to build essential skills by implementing a corner stone application of parallel computing which is The Game of Life.
In the project, we implement the Game of Life by using MPI(Message Passing Interface). We are asked to read an input file which constitutes the initial state of a grid of size 360x360 where each cell represents a state. The cells can take two values which are 0 and 1. The value of zero implies no existence, whereas the value of 1 implies a life. The state of the grid at time t depends on the state of the grid at time t-1. We are aiming at writing the final state of the grid into an output file after T iterations have been carried out by using parallel computing.

1.2 Solution

In the project, considering the parallel computing principles, the grid must be divided among the worker processes by the master process so that the worker processes can iterate their own sub portion.
The worker process share data among themselves in order to provide necessary information about the grid to compute the next state of the cells which they are responsible for. At this stage, given that the map is designed as a toroidal, and each process has 8 neighbors which it will communicate with, the data must be shared among processes accordingly. This version is called Periodic & Checkered version in the project description provided. We must also ensure that no deadlock occurs so that the program can execute successfully. Resolving the deadlock issues will be provided by a scheme designed during the implementation which will be explained in the Program Structure section in detail.
Given that we ensure no deadlock occurs during data share among processes, each worker process is ready to compute the next state of its sub portion. After the next state of each sub portion is calculated T times as descibed above, the worker processes resend their sub portions into the master process so that the master process can write the final state of the grid into the output file.

2 Program Interface

The project is implemented in Python (Python 3) by using the packages numpy and math, as well as mpi. To run the program, the user must ensure that these packages are available. After the user ensures that the requirements are satisfied, the following command in the terminal will run the program:

```
mpirun -np [M] --oversubscribe python3 2015400183.py [input.txt] [output.txt] [T]
```

In the command above, M stands for the number of processes that will run, T stands for the number of iterations. The input and output files will be given by the user as well.

3 Program Execution

The program can be executed via command-line interpreter. The user must declare the following arguments:

1. The number of processes that runs the program.
2. The input file which forms the initial state of the grid. The input file will contain 360 lines where each contains space-delimited cell values which are 0 or 1. The format supported is "txt" for the input file.
3. The output file is the file where the final state of the grid will be written into. The format supported is "txt" for the output file as well.
4. The number of iterations that the user wants must be given.

The arguments above will be provided as stated in Program Interface section.

4 Input and Output

The program takes a single input file to constitute the initial form of the grid. As stated in the project description previously, the size of the grid is 360x360. The input file must be in the format of "txt". The input file contains 360 lines where each line contains the values of the cells in a way that each value is separated by a single space. The initial values of cells can be 0 or 1. 0 implies no existence of a creature, and 1 implies a living creature.
The program outputs into a file which is declared by the user as parameter. The output file must be in "txt" file format as well. The file will be written in the same convention that is previously described for the input file. The file will contain exactly 360 non-empty lines where by the final states of the cells are separated by a single space.

5 Program Structure

5.1 Overall Structure

The structure of the game of life program can be regarded as two sub portions: The first is from the view of master process, and the other is from the view of worker processes. In order to facilitate parallel computing, we use mpi4py package which enables programmers to use Message-Passing Interface in pyhton. MPI provides useful variables that supplies the rank of the working process, the size of the communication world, and etc. After the initialization of MPI, the program will operate based on the rank number of the process: 0 for the master process, and nonzero ranks for the worker processes.
The master process reads the input file given as the parameter and constitutes a 2D array which represent the initial state of grid. This procedure will be conducted by the help of numpy package. Numpy package enables programmers to read the files into a numpy array by a single line instruction. The grid consists of binary values for each cell as described in Input and Output section. After the input file has been read, the master process assigns the equal sub portions of the grid into each worker process. At this stage, we must consider that the sub portions will be distributed according to the Checkered version where each worker process has 8 neighbors. Therefore we conclude that each worker process will take a sub portion of size (360/sqrt[N] * 360/sqrt[N]) from the grid where N denotes the number of worker processes. Calculation of the indexes that indicates the beginning and ending row and column numbers of a portion which a worker process is assigned will be carried out by the aid of the functions described in Functions subsection, since we need to calculate the row and column number of a process when the map is divided into [N] squared-areas to find the region where the process is responsible for. Therefore, the master process can successfully send the sub portions which the worker processes are assigned to.
After the master process sends the sub blocks of each worker process, we are ready to switch our perspective into the worker processes those having a nonzero rank. After each worker process received their sub blocks, we must consider data sharing among processes since they need the information about the cells in its boundary. Since these cells are in the region of its neighbors, a process must communicate with its neighbors to obtain necessary data to compute the next state of the cells that it's responsible for before starting into each iteration.
Before discussing about the deadlock avoidance mechanism that is followed during communication, let me introduce the functions that have been utilized in order to enable an easier implementation of communication which will later be provided in the Functions subsection as well. Recall that a process must communicate 8 of its neighbors which are located at north-west,north,north-east,west,east,south-west,south,south-east. In order to provide the rank of the process which we want to send information, we need supplementary functions that calculate the rank of the process that is located in a specific direction. Since we introduced the utilization of the supplementary functions, we are ready to describe the communication scheme that is used in the implementation.

In order to prevent the deadlocks, we must ensure a controlled communication scheme. At this stage we divide our worker processes according to their ranks. Odd-ranked workers send their data into the even-ranked workers while even-ranked workers wait for a receive, and vice versa. Notice that sending to bottom and up operations take place between two odd or even-ranked workers. Therefore, we can conclude that between odd-even communication provides every direction except the directions of topbottom. In order to provide a controlled communication plan to send/receive to/from topbottom, we should divide the worker processes in a way that they do not engage each other just as we did before by applying modulo according to the ranks. To provide this, we divide the worker processes according to their row indexes by modulo operation. Therefore, the odd-row indexed ones send into even row-indexed ones in the direction of up down, while even row-indexed ones wait for receive, and vice versa. This scheme allows us to implement communication without causing a deadlock.
Provided that the necessary data are shared among worker processes, it's time to carry out the game of life computations of each region by the worker processes according to the rules of the game of life. In order to provide an easier calculation step, each portion supplied by the neighbors of a worker process will constitute a new temporary 2D array with the sub portion that the worker process is already assigned. The data required by each neighbor will be placed into the appropriate locations in the temporary 2D array. Therefore we can say that, the new temporary array will be of size with 2 more rows and columns than the sub portion that the worker is already assigned. For a given cell, if the sum of the values of its neighbors is 3, then its next value is 1. If the sum is 2, then it retains its previous state. In other cases, it becomes a zero. After the next state has been calculated, then we are ready for the next iteration.
When the last iteration has been performed, the worker processes must return their sub portions into the master process which will reorganize the grid in order to provide the final state of its last form. The sub portions received will be replaced by the master process into the appropriate regions of the grid by using the supplementary functions again. After the reformation of the grid, the master process writes the content of the grid into the output file as described in Input and Output section. After the output file has been written, the program terminates.

5.2 Functions

5.2.1 Process Characteristics

This section includes the functions that returns the values about the necessary information about the worker processes.

1. get_row_num(rank_) and get_col_num(rank) functions return the row and column index values of a process. It must be stated that, the first index number of row and column for a worker process is 1.

2. get_process_id(row,col) function returns the rank of a process according

to its row and column indexes which actually reverts the functions stated above.

3. get_left(rank_), get_right(rank_), get_top(rank_), get_bottom(rank_), get_top_left(rank_), get_top_right(rank_), get_bottom_left(rank_), get_bottom_right(rank_) functions return the rank value of the neighbor process specified in a given direction for a worker process.

5.2.2 Communication

This section includes the functions that are used to provide a more simple communication scheme.

1. communicate_workers(map,cmd) function is used for master process to communicate all of worker processes either for a send or receive operation according to the value of the parameter cmd. The first argument is passed to reference the variable which holds the content of the grid.

2. send_neighbors(slice,cmd) function is used for worker processes to communicate each other. However, conversely, this function enables only one way of send functionality since receive requires many temporary variables which hold each data received from each neighbors. The functionality of this method depends on the second parameter cmd. If the cmd is same, then the send will take place between two odd or even-ranked worker processes. Else the method will carry out a send top & bottom operations.

6 Improvements and Extensions

In order to provide a better code writability and readibility, i have particularly been careful about the suitability for the coding principles of modularity and repetitions into loops. I noticed that since the receive from the neighbors requires 8 temporary variables that hold the values received from the neighbors, i did not put the receive instruction into a single function since it will require many parameters. However, a better design might be provided about this issue.

7 Difficulties Encountered

Previously, i have implemented game of life project using C++. However, the implementation by using Python was relatively much easier since the object manipulations are easier. The package numpy presents a much easier read/write from/to file operations as well. Therefore, thanks to the facilities given by Python, it was very easy to implement computations.
However, finding the appropriate communication scheme was hard, since the checkered version of the project requires that each worker process communicates with 8 of its neighbors. Considering the volume of the interactions for a given iteration, the deadlock may easily occur. However, as explained in Program Structure section, the communication scheme that i have chosen enabled an efficient implementation avoiding the deadlocks.

8 Conclusion

The game of life parallel programming project is very essential to build fundamental skills of modern computer science era. Parallel programming is one of the top trends which is profoundly becoming more popular in each day thanks to the breakthroughs achieved in computer design and hardware techniques. The programmers should utilize the facilities provided by the parallel programming in order to provide a better CPU utilization.
The project might be an essential start for the programmers which aims to build fundamental skills in parallel programming, since the project provides the main aspects of Message Passing Interface.

9 Appendices

This section includes the screenshots of the script.

```python
#Name: Baran Deniz Korkmaz
#Student ID:2015400183
#Compilation Status:Compiling
#Working Status:Working
#
#Implementation Details: Periodic & Checkered Version
#Run in Terminal: mpirun -np [n] --oversubscribe python3 2015400183.py [input.txt] [output.txt] [T]
#

#Below are the packages utilized for the project.
from mpi4py import MPI
import sys
import math
import numpy as np

#Defining some constant variables that are specified by the description.
MAP_SIZE=360
INPUT_FILE=sys.argv[1]
OUTPUT_FILE=sys.argv[2]
ITERATIONS=int(sys.argv[3])

#Initialize MPI environment and create global variables that will be used througout the program.
MASTER_PROCESS=0
comm=MPI.COMM_WORLD
rank=comm.Get_rank()
size=comm.Get_size()
num_of_workers=size-1
size_sqrt=int(math.sqrt(num_of_workers))

#get_row_num and get_col_num functions returns the row and column indexes of a given process id which is assigned to a specific part of the grid.
def get_row_num(rank_):
    return int(((rank_-1)/size_sqrt)+1)

def get_col_num(rank_):
    return (rank_-1)%size_sqrt+1

#Below are the functions which return the neighbor ids of a given process id which are assigned to the neighbor regions of that process.
def get_left(rank_):
    row_num_i=get_row_num(rank_)
    col_num_i=get_col_num(rank_)
    col_num_i=col_num_i - 1
    if(col_num_i < 1):
        col_num_i=col_num_i+size_sqrt
    return get_process_id(row_num_i,col_num_i)

def get_right(rank_):
    row_num_i = get_row_num(rank_)
```

```
46  def get_right(rank_):
47      row_num_i = get_row_num(rank_)
48      col_num_i = get_col_num(rank_)
49      col_num_i = col_num_i + 1
50      if (col_num_i > size_sqrt):
51          col_num_i = col_num_i - size_sqrt
52      return get_process_id(row_num_i, col_num_i)
53
54  def get_top(rank_):
55      row_num_i = get_row_num(rank_)
56      col_num_i = get_col_num(rank_)
57      row_num_i = row_num_i - 1
58      if (row_num_i < 1):
59          row_num_i = row_num_i + size_sqrt
60      return get_process_id(row_num_i, col_num_i)
61
62  def get_bottom(rank_):
63      row_num_i = get_row_num(rank_)
64      col_num_i = get_col_num(rank_)
65      row_num_i = row_num_i + 1
66      if (row_num_i > size_sqrt):
67          row_num_i = row_num_i - size_sqrt
68      return get_process_id(row_num_i, col_num_i)
69
70  def get_top_left(rank_):
71      return get_left(get_top(rank_))
72
73  def get_top_right(rank_):
74      return get_right(get_top(rank_))
75
76  def get_bottom_left(rank_):
77      return get_left(get_bottom(rank_))
78
79  def get_bottom_right(rank_):
80      return get_right(get_bottom(rank_))
81
82  #The id of a process can be calculated by using its row and column indexes. This function operates as a supplementer for the functions that are defined above.
83  def get_process_id(row,col):
84      return int((row-1)*size_sqrt+col)
85
86  #Master process communicates with all worker processes via this function. The function takes two arguments:
87  #The first argument is the 2D array which contains our grid.
88  #The second argument is a string which determines if the master will make a send or receive operation.
89  #In case of an invalid arg entry, the program returns the error code of -1.
90  def communicate_workers(map,cmd):
91      for i in range(1, num_of_workers + 1):
92          row_num = get_row_num(i)
```

```python
def communicate_workers(map,cmd):
    for i in range(1, num_of_workers + 1):
        row_num = get_row_num(i)
        col_num = get_col_num(i)
        row_from = int(MAP_SIZE / size_sqrt * (row_num - 1))
        row_to = int(MAP_SIZE / size_sqrt * (row_num))
        col_from = int(MAP_SIZE / size_sqrt * (col_num - 1))
        col_to = int(MAP_SIZE / size_sqrt * (col_num))
        if(cmd=="send"):
            temp_array = map[row_from:row_to, col_from:col_to]
            comm.send(temp_array, dest=i, tag=i)
        elif(cmd=="recv"):
            worker_result = comm.recv(source=i, tag=MASTER_PROCESS)
            map[row_from:row_to, col_from:col_to] = worker_result
        else:
            return(-1)

#The worker process communicates with its neighbors via this function. The function takes two arguments:
#The first argument is the 2D array which contains the subportion of the worker process.
#The second argument is a string which determines whether the communication process will be carried out with the neighbors that have the same modulo%2 value or n
#In case of an invalid arg entry, the program returns the error code of -1.
def send_neighbors(slice,cmd):
    if(cmd=="diff"):#Includes every direction except send top & bottom
        comm.send(slice[:, 0], dest=get_left(rank), tag=get_left(rank))  # send left
        comm.send(slice[:, len(slice) - 1], dest=get_right(rank), tag=get_right(rank))  # send right
        comm.send(slice[0, 0], dest=get_top_left(rank), tag=get_top_left(rank))  # send top left
        comm.send(slice[0, len(slice) - 1], dest=get_top_right(rank), tag=get_top_right(rank))  # send top right
        comm.send(slice[len(slice) - 1, 0], dest=get_bottom_left(rank), tag=get_bottom_left(rank))  # send bottom left
        comm.send(slice[len(slice) - 1, len(slice) - 1], dest=get_bottom_right(rank), tag=get_bottom_right(rank))  # send bottom right
    elif(cmd=="same"):#Includes send top & bottom
        comm.send(slice[0, :], dest=get_top(rank), tag=get_top(rank))  # send top
        comm.send(slice[len(slice) - 1, :], dest=get_bottom(rank), tag=get_bottom(rank))  # send bottom
    else:
        return(-1)

if __name__ == "__main__":
    if rank==0:#Master process operates inside of this block.
        #Master process read the input file and gets its content into a 2D array of size 360x360.
        map=np.genfromtxt(INPUT_FILE,delimiter=" ",dtype=int)

        #Master process sends the map portions into each processes which they should be assigned to.
        communicate_workers(map,"send")

        #Master process receives the final array subportions from the worker processes and inserts them into the proper places in the map.
        communicate_workers(map,"recv")

        #Since master has received the results and properly inserted them into the map, it finally writes the content into the output file.
```

```
        np.savetxt(fname=OUTPUT_FILE, X=map.astype(int), fmt='%.0f')

    else:#Worker processes operates inside of this block, since they have a nonzero id number.
        row_num_process = get_row_num(rank)
        col_num_process = get_col_num(rank)
        #Worker process receives the subportion from the 2D array of map and writes the content into the 2D array 'slice'.
        slice=comm.recv(source=MASTER_PROCESS,tag=rank)
        #slice_size=int(MAP_SIZE/size_sqrt)
        for i in range(ITERATIONS): #Iterations are implemented.
            #Odd-ranked processes carry out send&receive operations.
            if rank%2==1:
                #In order to prevent the deadlocks, the send&recv operations must be arranged.
                #Therefore in the case of odd-ranked processes, the process first sends into even numbered neighbors.
                send_neighbors(slice,"diff")
                # The processes with odd-rank & odd row indexes carry out send&receive to/from top/bottom processes so as to prevent deadlocks.
                if(row_num_process%2==1):
                    send_neighbors(slice,"same")
                    from_top=comm.recv(source=get_top(rank),tag=rank) #recv top
                    from_bottom=comm.recv(source=get_bottom(rank),tag=rank) #recv bottom
                else:# The processes with even row indexes carry out send&receive to/from top/bottom processes so as to prevent deadlocks.
                    from_top = comm.recv(source=get_top(rank), tag=rank)  # recv top
                    from_bottom = comm.recv(source=get_bottom(rank), tag=rank)  # recv bottom
                    send_neighbors(slice,"same")
                #The odd-ranked process now recieves from the even numbered neighbors.
                from_left=comm.recv(source=get_left(rank),tag=rank) #recv left
                from_right=comm.recv(source=get_right(rank),tag=rank) #recv right
                from_top_left=comm.recv(source=get_top_left(rank),tag=rank) #recv top left
                from_top_right=comm.recv(source=get_top_right(rank),tag=rank) #recv top right
                from_bottom_left=comm.recv(source=get_bottom_left(rank),tag=rank) #recv bottom left
                from_bottom_right=comm.recv(source=get_bottom_right(rank),tag=rank) #recv bottom right

            else:#Even-numbered processes carry out send&receive operations as explained above but in an opposite direction.
                from_left = comm.recv(source=get_left(rank), tag=rank)  # recv left
                from_right = comm.recv(source=get_right(rank), tag=rank)  # recv right
                from_top_left = comm.recv(source=get_top_left(rank), tag=rank)  # recv top left
                from_top_right = comm.recv(source=get_top_right(rank), tag=rank)  # recv top right
                from_bottom_left = comm.recv(source=get_bottom_left(rank), tag=rank)  # recv bottom left
                from_bottom_right = comm.recv(source=get_bottom_right(rank), tag=rank)  # recv bottom right

                if(row_num_process%2==1):
                    send_neighbors(slice,"same")
                    from_top=comm.recv(source=get_top(rank),tag=rank) #recv top
                    from_bottom=comm.recv(source=get_bottom(rank),tag=rank) #recv bottom
                else:
                    from_top = comm.recv(source=get_top(rank), tag=rank)  # recv top
                    from_bottom = comm.recv(source=get_bottom(rank), tag=rank)  # recv bottom
                    send_neighbors(slice,"same")
```

```
184
185                 send_neighbors(slice,"diff")
186
187             #In order to carry out game of life computations easier, the subportions received from the neighbors surrounds the slice of the worker process.
188             temp_array=np.zeros((len(slice)+2,len(slice)+2),dtype=int)
189             temp_array[1:len(slice)+1,1:len(slice)+1]=slice
190             temp_array[0,1:len(slice)+1]=from_top
191             temp_array[len(slice)+1,1:len(slice)+1]=from_bottom
192             temp_array[1:len(slice)+1,0]=from_left
193             temp_array[1:len(slice)+1,len(slice)+1]=from_right
194             temp_array[0,0]=from_top_left
195             temp_array[0,len(slice)+2-1]=from_top_right
196             temp_array[len(slice)+2-1,0]=from_bottom_left
197             temp_array[len(slice)+2-1,len(slice)+2-1]=from_bottom_right
198
199             #Game of life computations occur.
200             for i in range(1,len(slice)+1):
201                 for j in range(1,len(slice)+1):
202                     sum=int(temp_array[i-1,j-1])+int(temp_array[i-1,j])+int(temp_array[i-1,j+1])+int(temp_array[i,j-1])+int(temp_array[i,j+1])+int(temp_array[i+1
203                     if(sum<2 or sum>3):
204                         slice[i-1,j-1]=0
205                     else:
206                         if(sum==3):
207                             slice[i-1,j-1]=1 #The computations are stored in the slice array which is the permanent subportion of the process.
208
209         #Since all iterations have been done, the process sends its final subportion into the master process.
210         comm.send(slice,dest=MASTER_PROCESS,tag=MASTER_PROCESS)
```