BOGAZICI UNIVERSITY, ISTANBUL, TURKEY
CmpE300, Fall 2019

Project 1

Conway's Game of Life with MPI Programming Project

Bekir Yıldırım - 2014400054

22 December 2019

Date Performed .... 22 December, 2019
Instructor .... Tunga Gungor

Contents

1 Introduction

The game of life is a cellular automaton developed by the English mathematician Horton Conway in 1970. It is the best known example of cellular automaton. This game is actually an unmanned game, which means that the stages from human players are identified by the needless input states. It is that one influences the game of life by creating the initial configuration and observing how it develops. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent.
At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if by under population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

This document explains how we can implement the game of life algorithm using a parallel process and the challenges we face when implementing them. If I want to explain my method of solving problems in general, it is appropriate to mention that I implement the send and receive functions by performing a single double check without falling down the deadlock and find the processor's number at the corners of each cell easily. The details of my solution are explained in the following sections. In implementation, I choose periodic boundaries and checkered splits.

2 Program Interface

Before explaining implementation, we need to explain how the user will run the program or what requirements it must meet before running it. (Detailed within to macOS)

2.1 Before Running

- Launch a terminal session and run the following:

```
$  xcode-select --install
```

- Hit the install button (or get Xcode if you like).
- After the installation is complete, download Open MPI 4.0.2 from the link: https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.2.tar.gz
- Open up a terminal session and navigate inside the extracted Open MPI installation folder, called: openmpi-4.0.2 by default
- Run:

```
$  sudo ./configure --prefix=/usr/local
```

- Once it's done, run:

```
$  sudo make all install
```

- Above should complete your installation. Test it by running:

```
$  mpicc or $ mpirun
```

  without arguments.

- For python you should write also:

```
$  pip install mpi4py
```

- Program is terminated own by own.

2.2 The common issue

- To resolve issue of oversubscribe run the following command each time you start a new terminal session:

```
$ export OMPI_MCA_btl=self,tcp
```

2.3 While running

- To handle oversubscribe issue in python, open hostfile and write localhost slots=number of processors into it.

- Setting environment:

```
$  python3 -m venv env
$  source ./env/bin/activate
```

- Then, write below command to run:

```
$  mpiexec --hostfile hostfile -np 17 python mpi.py 3 rand_003.txt
```

  17 is proccesor number and python arguments are iteration, 3 = argv[1], and test file, rand_003.txt = argv[2].

3 Program Execution

The command line application takes 3 arguments. First one is the number of iteration and it's required. Second one is input file and it's required. Third one is test file and it's optional. If no file is provided, application prints out output to console. If file is provided, check with test file and write "Test case passed" to output.

3.1 Types of Inputs

Types of inputs are txt file and int.

3.2 Output of the Program

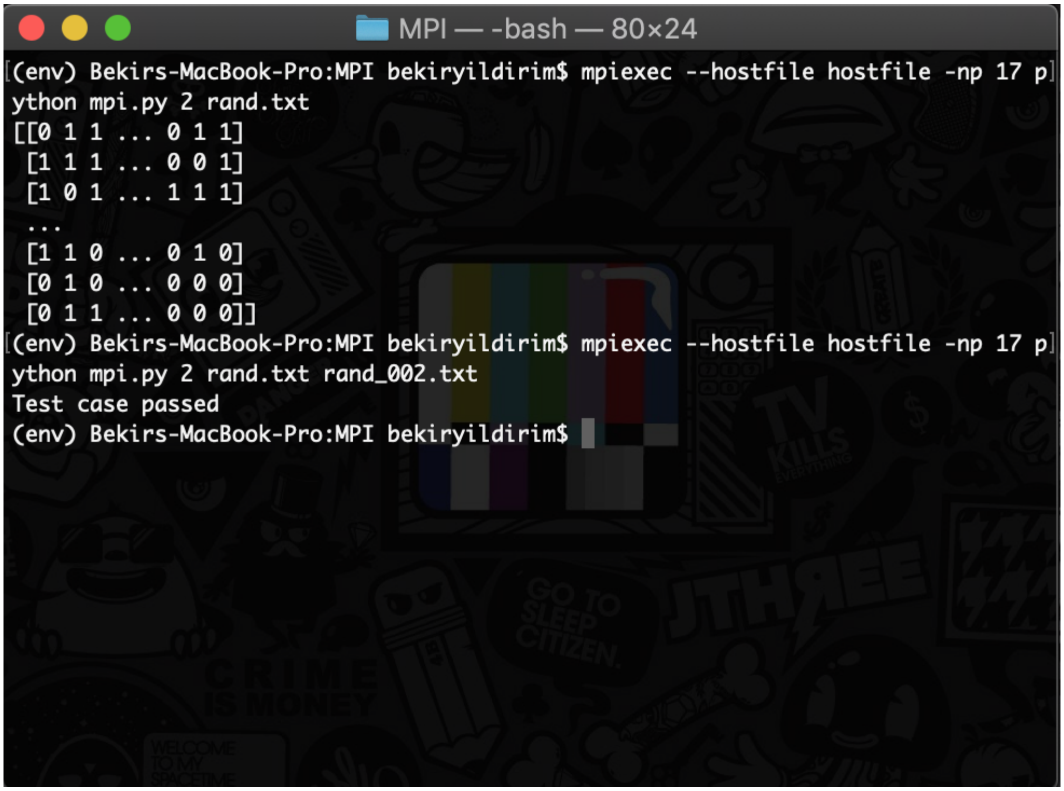


Figure 1: Program execution with given input.

4 Input & Output

4.1 Input

Since the size of the given inputs is too large to show in the pdf file, I will show an input with a smaller size.

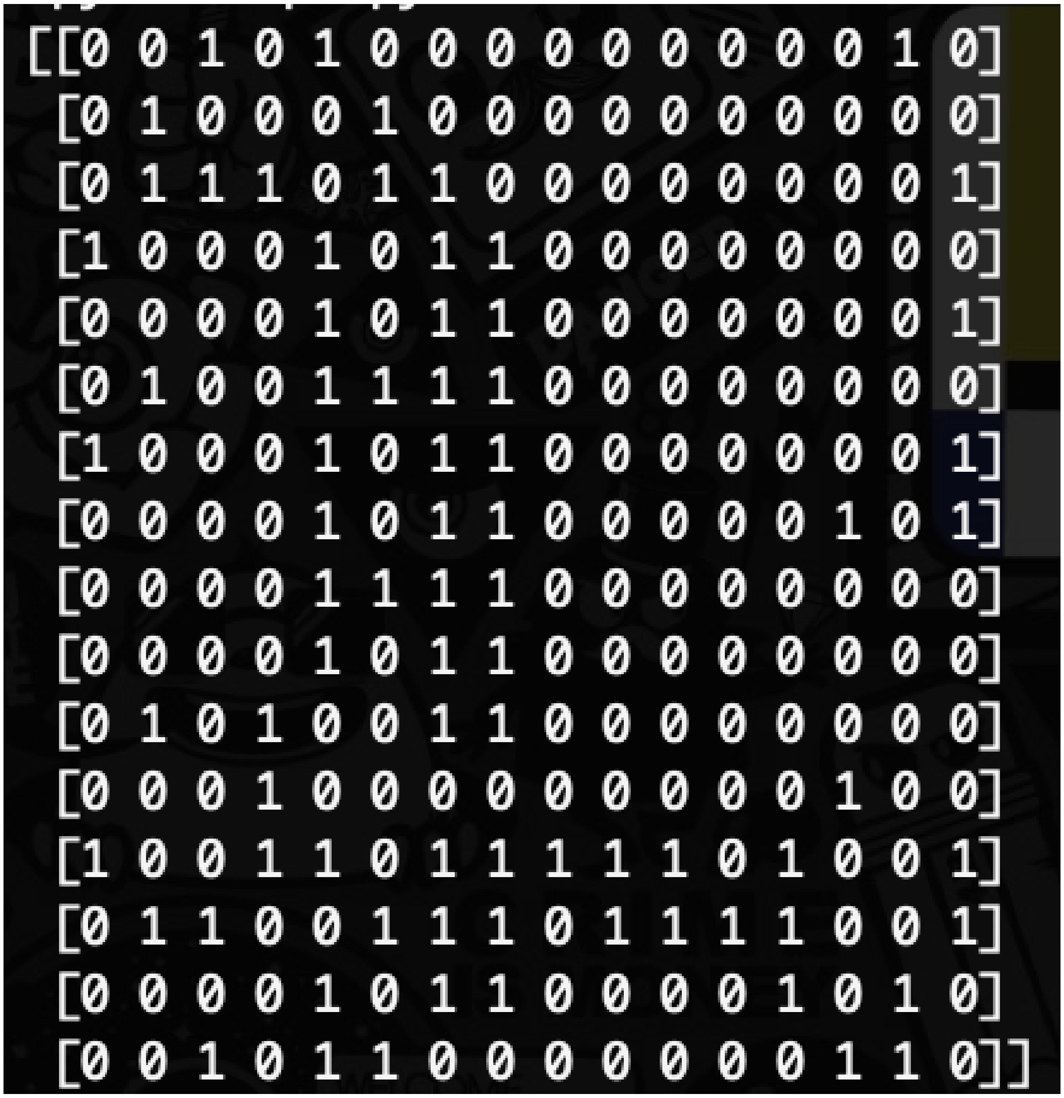
deneme.txt

```
1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
1 0 0 1 0 0 0 0 1 1 0 1 1 1 0 1
1 1 0 1 1 1 0 1 1 1 1 1 0 0 0 0
1 0 0 0 1 0 1 0 1 1 1 1 0 0 0 0
1 0 0 1 0 0 0 0 1 1 1 1 0 0 0 0
1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1
1 0 0 0 1 0 1 0 1 1 1 1 0 0 0 0
1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0
1 1 0 1 1 1 0 1 1 0 0 0 1 0 1 0
1 0 0 0 1 0 1 0 1 1 0 1 1 1 0 1
1 0 0 1 0 0 0 0 1 1 0 1 1 1 0 1
1 1 0 1 1 1 0 1 1 1 1 1 0 0 0 0
1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0
1 1 0 1 1 1 0 1 1 0 0 0 1 0 1 0
1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0
1 0 0 0 1 0 1 0 1 1 0 1 1 1 0 1
```

Figure 2: Sample Input

4.2 Output

The output is the version of the game of life algorithm made twice according to the result of the given sample input.



```
[[0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0]
 [0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 1 1 1 0 1 1 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 1]
 [0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0]
 [1 0 0 0 1 0 1 1 0 0 0 0 0 0 0 1]
 [0 0 0 0 1 0 1 1 0 0 0 0 0 1 0 1]
 [0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0]
 [0 1 0 1 0 0 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0]
 [1 0 0 1 1 0 1 1 1 1 1 0 1 0 0 1]
 [0 1 1 0 0 1 1 1 0 1 1 1 1 0 0 1]
 [0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 0]
 [0 0 1 0 1 1 0 0 0 0 0 0 0 1 1 0]]
```

Figure 3: Sample Output

5 Program Structure

This section describes the structure I used to solve the problem described in the introduction by dividing it into sub-sections.

5.1 Functions

5.1.1 find_top_below

Easily find each cell up or down. This function takes 4 parameters, world_rank which is hold rank of the processor, sqrt_slave which is hold the root of the number of slave processor, world_size which is hold the number of slave processor and control is the determine does a rank want the top one or the bottom one.

5.1.2 find_left_right

Easily find each cell left or right. This function takes 4 parameters, world_rank which is hold rank of the processor, sqrt_slave which is hold the root of the number of slave processor, world_size which is hold the number of slave processor and control is the determine does a rank want the left one or the right one.

5.1.3 send_ functions

The send function consists of a total of 8 functions and each performs the necessary operations for a different direction. This function takes 4 parameters, world_rank which is hold rank of the processor, sqrt_slave which is hold the root of the number of slave processor, world_size which is hold the number of slave processor and temp_array is the array sent from master processor to slave processor.

5.1.4 recv_ functions

The send function consists of a total of 8 functions and each performs the necessary operations for a different direction. This function takes 4 parameters, world_rank which is hold rank of the processor, sqrt_slave which is hold the root of the number of slave processor, world_size which is hold the number of slave processor.

5.1.5 neighbor_sum_local

This function summaries and returns the value of all cells in the 8 corners of a cell.

5.1.6 update_grid

This function traverses through all the cells of the given numpy array and recognizes whether each cell is subject to either OVERPOPULATION, LONELINESS or REPRODUCTION.

5.1.7 test

We put the numpy arrays that the last master process has collected and combined into a txt file and check that file with the required response. If the answer is consistent, we issue a "Test case passed" message.

5.2 Variables

This subsection section shows the variables I use to write more readable code and also reduce line counts

5.2.1 Global variables

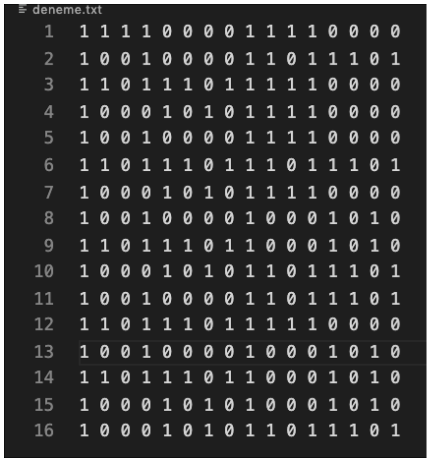
- MATRIX_SIZE : hold the matrix size
- FIND_TOP : hold the "up" to detect upper rank of the current rank
- FIND_BELOW : hold the "down" to detect upper rank of the current rank
- FIND_LEFT : hold the "left" to detect upper rank of the current rank
- FIND_RIGHT : hold the "right" to detect upper rank of the current rank
- LONELINESS : hold the "2" detect loneliness situation
- OVERPOPULATION : hold the "3" detect overpopulation situation
- REPRODUCTION : hold the "3" detect reproduction situation
- tag_ variables(UP, LEFT, etc.): used for reliable code.
- ITERATION : takes argument and determine iterate number.
- TEST_FILE : the file I compare my result to
- OUTPUT_FILE : the file I put my result
- comm : MPI instance

5.2.2 Local variables

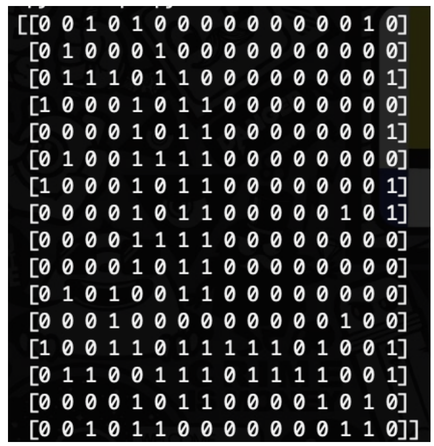
- world_size : hold the processor size - 1
- sqrt_slave : hold the root of the number of slave processor
- world_rank : hold the number of the current rank
- modulus_by_2 : hold the rank is even or odd
- modulus_by_sqrt_slave : hold the line of rank is even or odd
- length_slave_array : hold the length of split arrays sent by master process
- proc_grid : a variable that can hold arrays sent to corners from other rankings and open at the beginning of iteration to improve memory usage.
- temp_grid : hold the hard copy of "proc_grid" so that it doesn't always move through the changing array when the game of life algorithm is applied

6 Examples

The application executed with Figure 2.



(a) Sample Input.



(b) Sample Output.

Figure 4: Program Execution with Sample Input.

7 Improvements & Extensions

One of the problems I noticed when writing was that the master process was idle during the iteration and the slave processes were idle while distributing the master process array parts. At the same time, I realized that we could combine the parts without sending them to the master process. If we do this, I think the code will work in a more functional way.

8 Difficulties Encountered

One of the most difficult things in doing the project is to install mpi. Even though I made the directives in Pdf, there were some shortcomings and I had to research them online. I didn't have any difficulty implementing the project, but after writing the project, I encountered a lot of type errors when testing. Because type checkin is run-time in python, it can take a long time to find out where the error is.

9 Conclusions & Assessment

With this project, I had the opportunity to turn the theoretical knowledge that I had in mind about parallel programming into practice. I've seen more concrete examples of the deadlock incident. And as a result, I learned that when parallel programming, it is necessary to pay attention to transitions between processes, the order in which data is sent and received, or else the program will fall to deadlock.

10 Appendixes

```python
# Student Name: Bekir Yildirim
# Student Number: 2014400054
# Compile Status: Compiling
# Program Status: Working
# Periodic - Checkered

from sys import exit
import sys
from time import sleep
import numpy as np
from mpi4py import MPI
import math

MATRIX_SIZE = 360

FIND_TOP = "up"
FIND_BELOW = "down"
FIND_LEFT = "left"
FIND_RIGHT = "right"

# Game of life variables
LONELINESS = 2
OVERPOPULATION = 3
REPRODUCTION = 3

# Send-recv tag variables
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3
TOP_LEFT = 4
DOWN_RIGHT = 5
TOP_RIGHT = 6
DOWN_LEFT = 7

# iteration variable
ITERATION = sys.argv[1]

if len(sys.argv) > 3:
    TEST_FILE = sys.argv[3]

# output file variable
OUTPUT_FILE = 'output.txt'

comm = MPI.COMM_WORLD

# find top or below rank of current rank
def find_top_below(world_rank, sqrt_slave, world_size, control):
    return int((world_rank - sqrt_slave - 1) % world_size + 1) if control == FIND_TOP else int((world_rank + sqrt_slave - 1) % world_size + 1)

# find left or right rank of current rank
def find_left_right(world_rank, sqrt_slave, world_size, control):
    return int(int((world_rank - 1) / sqrt_slave) * sqrt_slave + (world_rank -2) % sqrt_slave + 1) if control == FIND_LEFT else int(int((world_rank - 1) / sqrt_slave) * sqrt_slave + (world_rank) % sqrt_slave + 1 )

# Send functions
```

```python
def send_right(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[:, -1], dest = find_left_right(world_rank, sqrt_slave, world_size, "right"), tag=RIGHT)

def send_left(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[:,0], dest=find_left_right(world_rank, sqrt_slave, world_size, "left"), tag=LEFT)

def send_down(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[-1,:], dest=find_top_below(world_rank, sqrt_slave, world_size, "down"), tag=DOWN)

def send_up(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[0,:], dest=find_top_below(world_rank, sqrt_slave, world_size, "up"), tag=UP)

def send_top_left(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[0,0], dest = find_left_right(find_top_below(world_rank, sqrt_slave, world_size, "up"), sqrt_slave, world_size, "left"), tag=TOP_LEFT)

def send_top_right(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[0,-1], dest = find_left_right(find_top_below(world_rank, sqrt_slave, world_size, "up"), sqrt_slave, world_size, "right"), tag=TOP_RIGHT)

def send_down_left(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[-1,0], dest = find_left_right(find_top_below(world_rank, sqrt_slave, world_size, "down"), sqrt_slave, world_size, "left"), tag=DOWN_LEFT)

def send_down_right(temp_array, world_rank, sqrt_slave, world_size):
    comm.send(temp_array[-1,-1], dest = find_left_right(find_top_below(world_rank, sqrt_slave, world_size, "down"), sqrt_slave, world_size, "right"), tag=DOWN_RIGHT)

# Recv functions
def recv_top(world_rank, sqrt_slave, world_size):
    return comm.recv(source = find_top_below(world_rank, sqrt_slave, world_size, "up"), tag=DOWN)

def recv_down(world_rank, sqrt_slave, world_size):
    return comm.recv(source = find_top_below(world_rank, sqrt_slave, world_size, "down"), tag=UP)

def recv_left(world_rank, sqrt_slave, world_size):
    return comm.recv(source = find_left_right(world_rank, sqrt_slave, world_size, "left"), tag=RIGHT)

def recv_right(world_rank, sqrt_slave, world_size):
    return comm.recv(source = find_left_right(world_rank, sqrt_slave, world_size, "right"), tag=LEFT)

def recv_top_left(world_rank, sqrt_slave, world_size):
    return comm.recv(source = find_left_right(find_top_below(world_rank, sqrt_slave, world_size, "up"), sqrt_slave, world_size, "left"), tag=DOWN_RIGHT)

def recv_top_right(world_rank, sqrt_slave, world_size):
```

```
    return comm.recv(source = find_left_right(find_top_below(world_rank,
    sqrt_slave, world_size, "up"), sqrt_slave, world_size, "right"), tag=
    DOWN_LEFT)

def recv_down_left(world_rank, sqrt_slave, world_size):
    return comm.recv(source =  find_left_right(find_top_below(world_rank,
    sqrt_slave, world_size, "down"), sqrt_slave, world_size, "left"), tag=
    TOP_RIGHT)

def recv_down_right(world_rank, sqrt_slave, world_size):
    return comm.recv(source = find_left_right(find_top_below(world_rank,
    sqrt_slave, world_size, "down"), sqrt_slave, world_size, "right"), tag=
    TOP_LEFT)

# look all neighbor of cell and sum them.
def neighbor_sum_local(proc_grid,row,col):
    return proc_grid[row+1,col+1] + proc_grid[row+1,col] + proc_grid[row+1,
    col-1] + \
      proc_grid[row,col+1] + proc_grid[row,col-1] + \
      proc_grid[row-1,col+1] + proc_grid[row-1,col] + proc_grid[row-1,col-1]

# provide game of life algorithm with traversing array.
def update_grid(temp_grid, proc_grid):
    number_of_iterate = len(proc_grid)
    for row in range(1,number_of_iterate-1):
        for col in range(1, number_of_iterate-1):
            if proc_grid[row,col] and  neighbor_sum_local(proc_grid, row, col
    ) > OVERPOPULATION:
                temp_grid[row,col] = 0
            elif proc_grid[row,col] and neighbor_sum_local(proc_grid, row,
    col) < LONELINESS:
                temp_grid[row,col] = 0
            elif neighbor_sum_local(proc_grid, row, col) == REPRODUCTION:
                temp_grid[row,col] = 1
    return temp_grid

# test output with test file
def test(input_array):
    np.savetxt(OUTPUT_FILE, input_array.astype(int), fmt = '%i')
    myResult = np.loadtxt(OUTPUT_FILE, dtype=int)
    forTest = A = np.loadtxt(TEST_FILE, dtype=int)
    if(np.array_equal(myResult, forTest)):
        print("Test case passed")

#   Main function; master, which has rank 0, reads the input file and shares
    them accross to slaves.
#   Then slaves do some calculations to do game of life, then slaves sends it
     to master process. Then master merges and prints the result to output
    file.
if __name__ == "__main__":

    # Initializes the MPI.
    world_size = comm.Get_size()-1
    sqrt_slave = int(math.sqrt(world_size))
    world_rank = comm.Get_rank()


    # If it is master process, reads the input from file. And shares them to
    slaves.
```

```
# Then it merges results of slaves' operations. Then prints the result to the output file.
if world_rank == 0:

    input_array = np.genfromtxt(sys.argv[2], delimiter=" ", dtype=int)  # Reads input file.
    rows_per_slave = int(MATRIX_SIZE/math.sqrt(world_size))
    rank_count = 1
    # Shares input to the slaves.
    for i in range(int(math.sqrt(world_size))):
        for j in range(int(math.sqrt(world_size))):
            comm.send(input_array[i*rows_per_slave:i*rows_per_slave+rows_per_slave,
                                  j*rows_per_slave:j*rows_per_slave+rows_per_slave], dest=rank_count, tag=0)
            rank_count += 1

    rank_count = 1

    # Receives result of slaves' operations and merges them.
    for i in range(int(math.sqrt(world_size))):
        for j in range(int(math.sqrt(world_size))):
            input_array[i*rows_per_slave:i*rows_per_slave+rows_per_slave,
                                  j*rows_per_slave:j*rows_per_slave+rows_per_slave] = comm.recv(source=rank_count, tag=rank_count)
            rank_count += 1

    # Prints the result to the output file or console.
    if len(sys.argv) > 3:
        test(input_array)
    else :
        print(input_array)

# Slaves function; gets own matrix from master. Then does 'ITERATION' executions to do game of life
# with communicating with other slaves. Then sends result to the master processor.
else:
    modulus_by_2 = world_rank % 2
    modulus_by_sqrt_slave = int(world_rank/sqrt_slave) % 2
    temp_array = comm.recv(source=0, tag=0)
    length_slave_array = len(temp_array)
    # 'proc_grid' a variable that can hold arrays sent to corners from other rankings andopen at the beginning of iteration to improve memory usage.
    proc_grid = np.zeros((length_slave_array + 2, length_slave_array + 2), dtype=int)
    for i in range(int(ITERATION)):
        proc_grid[1:length_slave_array+1,1:length_slave_array+1] = temp_array

        # determine whether rank is even or odd.
        if modulus_by_2 == 0:

            # recv of even rank
            proc_grid[1:length_slave_array+1,0] = recv_left(world_rank, sqrt_slave, world_size)
            proc_grid[1:length_slave_array+1,-1] = recv_right(world_rank, sqrt_slave, world_size)
```

```python
            proc_grid[0,0] = recv_top_left(world_rank, sqrt_slave, world_size)
            proc_grid[0,-1] = recv_top_right( world_rank, sqrt_slave, world_size)
            proc_grid[-1,0] = recv_down_left( world_rank, sqrt_slave, world_size)
            proc_grid[-1,-1] = recv_down_right(world_rank, sqrt_slave, world_size)

            # determine whether line of rank is even or odd.
            if modulus_by_sqrt_slave == 0:
                send_down(temp_array, world_rank, sqrt_slave, world_size)
                send_up(temp_array, world_rank, sqrt_slave, world_size)
                proc_grid[0,1:length_slave_array+1] = recv_top(world_rank, sqrt_slave, world_size)
                proc_grid[-1,1:length_slave_array+1] = recv_down(world_rank, sqrt_slave, world_size)

            else :
                proc_grid[0,1:length_slave_array+1] = recv_top(world_rank, sqrt_slave, world_size)
                proc_grid[-1,1:length_slave_array+1] = recv_down(world_rank, sqrt_slave, world_size)
                send_up(temp_array, world_rank, sqrt_slave, world_size)
                send_down(temp_array, world_rank, sqrt_slave, world_size)

            # send right part
            send_right(temp_array, world_rank, sqrt_slave, world_size)
            # send top left
            send_left(temp_array, world_rank, sqrt_slave, world_size)
            # send below left
            send_down_left(temp_array, world_rank, sqrt_slave, world_size)
            # send below right
            send_down_right(temp_array, world_rank, sqrt_slave, world_size)
            # send top right
            send_top_right(temp_array, world_rank, sqrt_slave, world_size)
            # send left part
            send_top_left(temp_array, world_rank, sqrt_slave, world_size)

        elif modulus_by_2 == 1:

            # send right part
            send_right(temp_array, world_rank, sqrt_slave, world_size)
            # send top left
            send_left(temp_array, world_rank, sqrt_slave, world_size)
            # send below right
            send_down_right(temp_array, world_rank, sqrt_slave, world_size)
            # send below left
            send_down_left(temp_array, world_rank, sqrt_slave, world_size)
            # send top right
            send_top_right(temp_array, world_rank, sqrt_slave, world_size)
            # send left part
            send_top_left(temp_array, world_rank, sqrt_slave, world_size)

```

```
                # determine whether line of rank is even or odd.
                if modulus_by_sqrt_slave == 0:
                    send_down(temp_array, world_rank, sqrt_slave, world_size)
                    send_up(temp_array, world_rank, sqrt_slave, world_size)
                    proc_grid[0,1:length_slave_array+1] = recv_top(world_rank
, sqrt_slave, world_size)
                    proc_grid[-1,1:length_slave_array+1] = recv_down(
world_rank, sqrt_slave, world_size)

                else :
                    proc_grid[0,1:length_slave_array+1] = recv_top(world_rank
, sqrt_slave, world_size)
                    proc_grid[-1,1:length_slave_array+1] = recv_down(
world_rank, sqrt_slave, world_size)
                    send_up(temp_array, world_rank, sqrt_slave, world_size)
                    send_down(temp_array, world_rank, sqrt_slave, world_size)

                # recv of odd rank
                proc_grid[1:length_slave_array+1,0] = recv_left(world_rank,
sqrt_slave, world_size)
                proc_grid[1:length_slave_array+1,-1] = recv_right(world_rank,
 sqrt_slave, world_size)
                proc_grid[0,0] = recv_top_left(world_rank, sqrt_slave,
world_size)
                proc_grid[0,-1] = recv_top_right(world_rank, sqrt_slave,
world_size)
                proc_grid[-1,0] = recv_down_left(world_rank, sqrt_slave,
world_size)
                proc_grid[-1,-1] = recv_down_right(world_rank, sqrt_slave,
world_size)

            # 'temp_grid' hold the hard copy of     procgrid     so that it
 doesnt always move throughthe changing array when the game of life
algorithm is applied
            temp_grid = np.copy(proc_grid)
            proc_grid = update_grid(temp_grid, proc_grid)

            temp_array = proc_grid[1:length_slave_array+1,1:
length_slave_array+1]

        # Send the result to the master processor.
        comm.send(temp_array, dest = 0, tag=world_rank)

    # Finalizes the MPI and process.
    MPI.Finalize()
```