TESTING GAME OF LIFE PROGRAM ON SUPERCOMPUTER

Homework # 2

By

Yichen Huang

CS 481 High-Performance Computing

September 10, 2021

I. Problem Specification

The Game of Life is devised by British mathematician John Horton Conway in 1970 [1]. The Game of Life is a board game that consists of a 2-dimensional array of cells. There are only two statuses for each cell – dead or alive. The games start with an initial state and the cells either die, live or multiply in the next generation with rules. Each cell will have eight neighbors – top left, top, top right, left, right, bottom left, bottom, and bottom right.

If a cell is alive in the current generation, then the cell will either live or dead in the next generation depend on the states of its neighbor with the following rules:

- If the cell has less than one alive neighbor, then the cell will die in the next generation because of loneliness.
- If the cell has more than four alive neighbors, then the cell will die in the next generation because of overpopulation.
- If the cell has exactly two or three alive neighbors, then the cell will survive in the next generation.

If a cell is dead in the current generation, and if there are exactly three neighbors alive then the cell will be changed to alive in the next generation due to neighboring cells give birth to a new organism. All cells in this game are affected simultaneously in one generation. The game will end when no change between two generations or reach the maximum generation count.

In this assignment, the code will be stored in the GitHub personal repository and tested on the UA Supercomputer Center.

II. Program Design

The program will be implemented by C with two inputs in command line execution – the size of the board(N) and the maximum generation number(max_gen). The program will exit with -1 when the size of the board and maximum generation number are not included in the command line.

The program will allocate two arrays – life1 and life2 – dynamically to store the status of each cell. The size of each array will be (N+2) x (N+2). There will be a layer of ghost cells around the board which will be set to death (0) permanently.

To generate the initial generation, the program will employ randomize method to the life1 array. If the cell is alive in the original generation, it will be set to 1. If the cell is dead in the initial generation, it will be set to 0.

During computation, a pointer will iterate every cell in life1 and compute the number of living cells around that cell. With the rules specified in section II, the status in the next generation will be saved in the life2 array with the same row and column. During the process, a flag will record if there are differences between the two graphs. After iterating all cells, the new status of the next generation will be saved in the life2 array. In the final step, if the flag shows that there is no difference between the two graphs, the program will stop. If the two graphs are different, the life1 pointer and life2 pointer will be switched which means life1 will become the latest version.

At end of the program, life1 and life2 will be manually released. The program will show the time cost for the processing.

A private repository in the GitHub was created to store and keep update record for all the source code. The output of test cases will also be store in the repository.

III. Testing Plan

The program will apply a random number generator to initialize the board of the original generation. The testing plan will have two parts.

The first part is to test the program with a small problem size (3x3, 5x5, or 10x10). Each generation will be printed out which can check if the program runs properly.

Then after testing the small size problem, in the second part, the large size problem will be tested. The program will not print out each generation because of the size of the problem; however, the time taken will be printed at end of the program. Each problem will be run three-time and use the average as the time taken.

In Homework#1, the tests are set on local machine. However, in this assignment, the program will be tested on UA Supercomputer Center. All tests will be running on one core in intel CPU and on DMC cluster. For size 1000*1000 problem, the program will be given 1 GB memory. For size 5000*5000 problem, the program will be given 2 GB memory. For size 10,000*10,000 problem, the program will be given 8 GB memory. All The results of the tests will be compared with the result from local machine.

IV. Test Cases

The result of part I of the testing plan will be shown in Appendix I. The results of part II are shown in Table I.

Test Case #	Problem Size	Max. Generations	Time Taken Homework-1(s)	Time Taken DMC cluster(s)
1	1000x1000	1000	11.62	17.18
2	1000x1000	5000	54.42	81.81
3	5000x5000	1000	287.88	467.20
4	5000x5000	5000	1481.12	2193.95
5	10000x10000	1000	1345.14	1854.79
6	10000x10000	5000	6665.74	8284.57

Table I. The time usage of large test case[1]

[1] The random generator is set with srand(2)

V. Analysis and Conclusion

The local machine used to run the program is the 2015 MacBook Pro. The operating system is macOS Catalina (Version 10.15.7). The CPU of the machine is Intel® Core™ i7-4870HQ, the clock speed of the CPU is 2.50 GHz. The memory in the system is 16 GB.

The program is coded in C, the compiler is gcc with -std=c99 and -Wall. Because of the difference of random number generator in MacOS and Linux system. The random number generator was changed from rand() to random() in MacOS in order to make sure the test cases in Linux and MacOS are same.

Because the program will iterate a two-dimensional array, the time complexity should be $O(N*N^2)$ which in general is $O(N^2)$. The results from the Part II of the test case also match the prediction that when generation is same, the time taken increase about 25 times for problem size increase from 1000 to 5000; and when problem size is constant, the time taken increase about 5 times for maximum generation increase from 1000 to 5000.

From the results in Table I, for the same test case, the supercomputer took more time than local computer. There exists reason can explain this situation. The CPU used in supercomputer center is probably old and the base frequency of single core is less than that of local machine. Because of the frequency decide the computational speed, the time taken on supercomputer is greater than local machine.

The difficult part of this assignment is to reduce the time consumption when running a large-size problem. In the first attempt, to exchange life1 and life2, the program iterates each element of two arrays and exchange each element which will cost more time than just exchange the pointers of two arrays. After applicating exchange pointers instead of deep copy, the time taken for large problems reduces about 15%.

To further improve the program, parallel programming is a good method to reduce the time taken for the large-size problem. The CPU used in local machine has 4 cores and more cores can be requested in supercomputer which if separate the task into 4 cores can have a huge decrease in time consumption.

Appendix I

The results from local machine are same as results in supercomputer.

- srand(0)
  - size 3, max_gen 10

```
In Gen# 0
1 0 1
1 1 1
0 0 1

In Gen# 1
1 0 1
1 0 1
0 0 1

In Gen# 2
0 0 0
0 0 1
0 1 0

In Gen# 3
0 0 0
0 0 0
0 0 0
```

  - size 5, max_gen 10

```
In Gen# 0
1 0 1 1 1
1 0 0 1 1
0 1 0 1 1
0 0 0 0 0
1 0 1 1 0

In Gen# 1
0 1 1 0 1
1 0 0 0 0
0 0 1 1 1
0 1 0 0 1
0 0 0 0 0

In Gen# 2
0 1 0 0 0
0 0 0 0 1
0 1 1 1 1
0 0 1 0 1
0 0 0 0 0

In Gen# 3
0 0 0 0 0
0 1 0 0 1
0 1 1 0 1
0 1 1 0 1
0 0 0 0 0
```

```
In Gen# 4
0 0 0 0 0
0 1 1 1 0
1 0 0 0 1
0 1 1 0 0
0 0 0 0 0


In Gen# 5
0 0 1 0 0
0 1 1 1 0
1 0 0 0 0
0 1 0 0 0
0 0 0 0 0

In Gen# 6
0 1 1 1 0
0 1 1 1 0
1 0 0 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 7
0 1 0 1 0
1 0 0 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 8
0 0 1 0 0
1 0 0 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 9
0 0 0 0 0
0 0 0 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 10
0 0 0 0 0
0 0 1 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
```

- size 10, max_gen 10

```
In Gen# 0
1 0 1 1 1 1 0 0 1 1
0 1 0 1 1 0 0 0 0 0
1 0 1 1 0 0 0 1 1 1
1 0 0 0 1 1 1 0 1 0
1 1 1 1 0 1 0 0 1 0
1 0 1 0 0 1 0 0 0 1
1 1 0 1 0 1 0 1 1 1
0 1 0 1 0 1 0 0 1 0
1 0 0 0 0 0 1 1 0 1
0 0 0 0 1 0 0 0 0 1

In Gen# 1
0 1 1 0 0 1 0 0 0 0
1 0 0 0 0 1 0 1 0 0
1 0 1 0 0 0 1 1 1 1
1 0 0 0 0 1 1 0 0 0
1 0 1 1 0 0 0 1 1 1
0 0 0 0 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1
0 1 0 0 0 1 0 0 0 0
0 0 0 0 1 1 1 1 0 1
0 0 0 0 0 0 0 0 1 0

In Gen# 2
0 1 0 0 0 0 1 0 0 0
1 0 1 0 0 1 0 1 0 0
1 0 0 0 0 0 0 0 1 0
1 0 1 1 0 1 0 0 0 0
0 1 0 0 1 1 0 1 0 1
0 1 1 1 0 0 0 1 0 1
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 1 1 1 1 0

In Gen# 3
0 1 0 0 0 0 1 0 0 0
1 0 0 0 0 0 1 1 0 0
1 0 1 1 1 0 1 0 0 0
1 0 1 1 0 1 1 0 1 0
1 0 0 0 0 1 0 0 0 0
0 1 1 1 0 1 0 0 0 0
0 0 1 0 0 0 1 0 1 0
0 0 0 0 1 0 0 1 1 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 1 0
```

```
In Gen# 4
0 0 0 0 0 0 1 1 0 0
1 0 1 1 0 0 1 1 0 0
1 0 1 0 1 0 0 0 0 0
1 0 1 0 0 0 1 1 0 0
1 0 0 0 0 1 0 0 0 0
0 1 1 1 1 1 1 0 0 0
0 1 1 0 1 1 1 0 1 0
0 0 0 1 0 1 0 1 1 0
0 0 0 1 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0

In Gen# 5
0 0 0 0 0 0 1 1 0 0
0 0 1 1 0 1 1 1 0 0
1 0 1 0 0 1 0 0 0 0
1 0 0 1 0 1 1 0 0 0
1 0 0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 1 0 1 1 0
0 0 0 0 1 0 0 0 0 0

In Gen# 6
0 0 0 0 0 1 0 1 0 0
0 1 1 1 1 1 0 1 0 0
0 0 1 0 0 0 0 1 0 0
1 0 0 0 1 1 1 0 0 0
1 1 0 0 0 0 0 1 0 0
1 1 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 1
0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 0 0 0 0 0

In Gen# 7
0 0 1 1 0 1 0 0 0 0
0 1 1 1 1 1 0 1 1 0
0 0 1 0 0 0 0 1 0 0
1 0 0 0 0 1 1 1 0 0
0 0 0 0 0 1 0 1 1 0
1 1 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0
```

```
In Gen# 8
0 1 0 0 0 1 1 0 0 0
0 1 0 0 0 1 0 1 1 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
1 1 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0

In Gen# 9
0 0 0 0 0 1 1 1 0 0
0 1 1 0 0 1 0 1 0 0
0 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

In Gen# 10
0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0 0
0 1 1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

- srand(1)
    - size 3, max_gen 10

```
In Gen# 0
1 0 1
1 1 1
0 0 1

In Gen# 1
1 0 1
1 0 1
0 0 1
```

```
In Gen# 2
0 0 0
0 0 1
0 1 0

In Gen# 3
0 0 0
0 0 0
0 0 0
```

- size 5, max_gen 10

```
In Gen# 0
1 0 1 1 1
1 0 0 1 1
0 1 0 1 1
0 0 0 0 0
1 0 1 1 0

In Gen# 1
0 1 1 0 1
1 0 0 0 0
0 0 1 1 1
0 1 0 0 1
0 0 0 0 0

In Gen# 2
0 1 0 0 0
0 0 0 0 1
0 1 1 1 1
0 0 1 0 1
0 0 0 0 0

In Gen# 3
0 0 0 0 0
0 1 0 0 1
0 1 1 0 1
0 1 1 0 1
0 0 0 0 0

In Gen# 4
0 0 0 0 0
0 1 1 1 0
1 0 0 0 1
0 1 1 0 0
0 0 0 0 0

In Gen# 5
0 0 1 0 0
0 1 1 1 0
1 0 0 0 0
0 1 0 0 0
0 0 0 0 0
```

```
In Gen# 6
0 1 1 1 0
0 1 1 1 0
1 0 0 0 0
0 0 0 0 0
0 0 0 0 0



In Gen# 7
0 1 0 1 0
1 0 0 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 8
0 0 1 0 0
1 0 0 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 9
0 0 0 0 0
0 0 0 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

In Gen# 10
0 0 0 0 0
0 0 1 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
```

- size 10, max_gen 10

```
In Gen# 0
1 0 1 1 1 1 0 0 1 1
0 1 0 1 1 0 0 0 0 0
1 0 1 1 0 0 0 1 1 1
1 0 0 0 1 1 1 0 1 0
1 1 1 1 0 1 0 0 1 0
1 0 1 0 0 1 0 0 0 1
1 1 0 1 0 1 0 1 1 1
0 1 0 1 0 1 0 0 1 0
1 0 0 0 0 0 1 1 0 1
0 0 0 0 1 0 0 0 0 1
```

```
In Gen# 1
0 1 1 0 0 1 0 0 0 0
1 0 0 0 0 1 0 1 0 0
1 0 1 0 0 0 1 1 1 1
1 0 0 0 0 1 1 0 0 0
1 0 1 1 0 0 0 1 1 1
0 0 0 0 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1
0 1 0 0 0 1 0 0 0 0
0 0 0 0 1 1 1 1 0 1
0 0 0 0 0 0 0 0 1 0



In Gen# 2
0 1 0 0 0 0 1 0 0 0
1 0 1 0 0 1 0 1 0 0
1 0 0 0 0 0 0 0 1 0
1 0 1 1 0 1 0 0 0 0
0 1 0 0 1 1 0 1 0 1
0 1 1 1 0 0 0 1 0 1
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 1 1 1 1 0

In Gen# 3
0 1 0 0 0 0 1 0 0 0
1 0 0 0 0 0 1 1 0 0
1 0 1 1 1 0 1 0 0 0
1 0 1 1 0 1 1 0 1 0
1 0 0 0 0 1 0 0 0 0
0 1 1 1 0 1 0 0 0 0
0 0 1 0 0 0 1 0 1 0
0 0 0 0 1 0 0 1 1 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 1 0

In Gen# 4
0 0 0 0 0 0 1 1 0 0
1 0 1 1 0 0 1 1 0 0
1 0 1 0 1 0 0 0 0 0
1 0 1 0 0 0 1 1 0 0
1 0 0 0 0 1 0 0 0 0
0 1 1 1 1 1 1 0 0 0
0 1 1 0 1 1 1 0 1 0
0 0 0 1 0 1 0 1 1 0
0 0 0 1 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0
```

```
In Gen# 5
0 0 0 0 0 0 1 1 0 0
0 0 1 1 0 1 1 1 0 0
1 0 1 0 0 1 0 0 0 0
1 0 0 1 0 1 1 0 0 0
1 0 0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 1 0 1 1 0
0 0 0 0 1 0 0 0 0 0

In Gen# 6
0 0 0 0 0 1 0 1 0 0
0 1 1 1 1 1 0 1 0 0
0 0 1 0 0 0 0 1 0 0
1 0 0 0 1 1 1 0 0 0
1 1 0 0 0 0 0 1 0 0
1 1 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 1
0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 0 0 0 0 0

In Gen# 7
0 0 1 1 0 1 0 0 0 0
0 1 1 1 1 1 0 1 1 0
0 0 1 0 0 0 0 1 0 0
1 0 0 0 0 1 1 1 0 0
0 0 0 0 0 1 0 1 1 0
1 1 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0

In Gen# 8
0 1 0 0 0 1 1 0 0 0
0 1 0 0 0 1 0 1 1 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
1 1 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0
```

```
In Gen# 9
0 0 0 0 0 1 1 1 0 0
0 1 1 0 0 1 0 1 0 0
0 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

In Gen# 10
0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0 0
0 1 1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Reference

[1] M. Gardner, "MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life"," October 1970. [Online]. Available: https://web.stanford.edu/class/sts145/Library/life.pdf. [Accessed 25 August 2021].

[2] The allocateArray(int **array, int size) in line 25, freeArray(int **array) in line 51, and gettime() in line56 functions in the code is modifed by hw1.c from blackboard Homework#1.