DESIGN AND IMPLEMENT MULTITHREADED VERSION OF GAME OF LIFE PROGRAM WITH OPENMP

Homework # 3

By

Yichen Huang

CS 481 High-Performance Computing

October 9, 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 multithread API – OpenMP will be employed to increase the performance of the program in assignment#1 with a multithread method.

II. Program Design

The program is modified from the solution of assignment#1 – life.c – from the blackboard. The program has four inputs in command line execution – the size of the board(N), maximum generation number(max_gen), number of thread(num_threads), and output directory(output_dir). The program will exit with -1 when there are not enough command line arguments. The output file will be store in the local directory if the program is running in a local machine. When testing on the ASC cluster, the output file will be stored in/scratch/$USER directory.

The program will allocate two arrays – life and temp – 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 achieve a multithreaded program, the program will create threads before the outer loop, and apply “omp for” when iterating the array and computing the next generation. Each thread will be arranged *(Size^2)/num_threads* cells which can increase the computational speed. After calculation of all cells. To improve the performance, a single thread will swap two pointers

instead of a master thread. After swapping, the flag which records if there exists differences between two generations will determine if to stop the whole loop.

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 the life array 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 temp 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 temp array. In the final step, if the flag shows that there is no difference between the two graphs with is equal to 0, the loop will stop. If the two graphs are different, the life pointer and temp pointer will be switched which means the life array will become the latest version.

At end of the program, life and temp pointers will be manually released. The program will show the time cost for the processing. The result of the life array will be print to the output file and store in the directory from the input.

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 check if the output of the multithread version is correct from the sequential version. Part one will check the output from the multithread version program and sequential program from blackboard with1000x1000 size of the game with 100 iterations and 5000 iterations. The multithread program will be run with thread numbers of 1, 2, 4, 8, 10, 16, and 20. The “diff” command will be used to check the difference between output files. The output file of sequential program will be named as ans.<size>.<max_gen>.1 and the output file of multithread program will be named as output.<size>.<max_gen>.<thread_number>.

The second part is to test the performance difference between the different number of threads. This part will test the multithread version of the program with size as 5000, max_gen as 5000, and thread number with 1, 2, 4, 8, 10, 16, and 20. With three runs, the average time taken will be recorded.

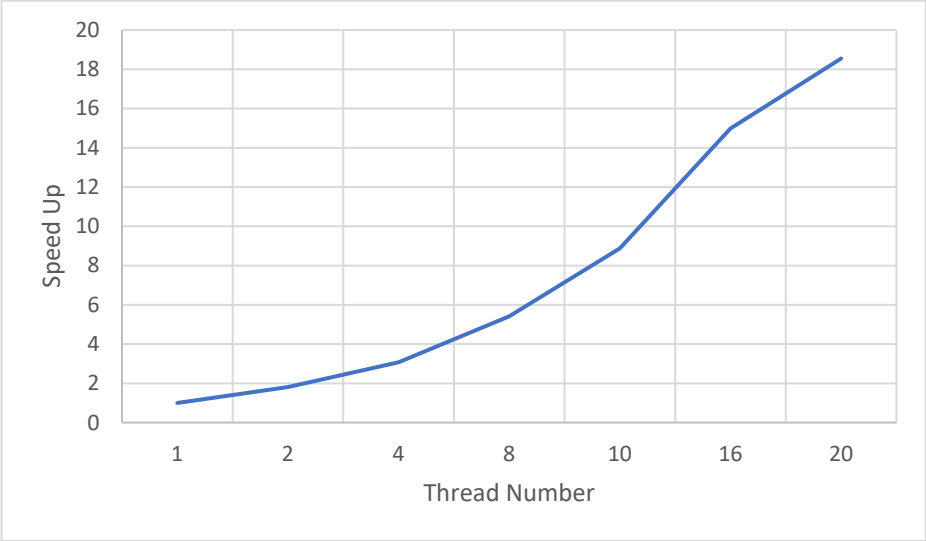
IV. Test Cases

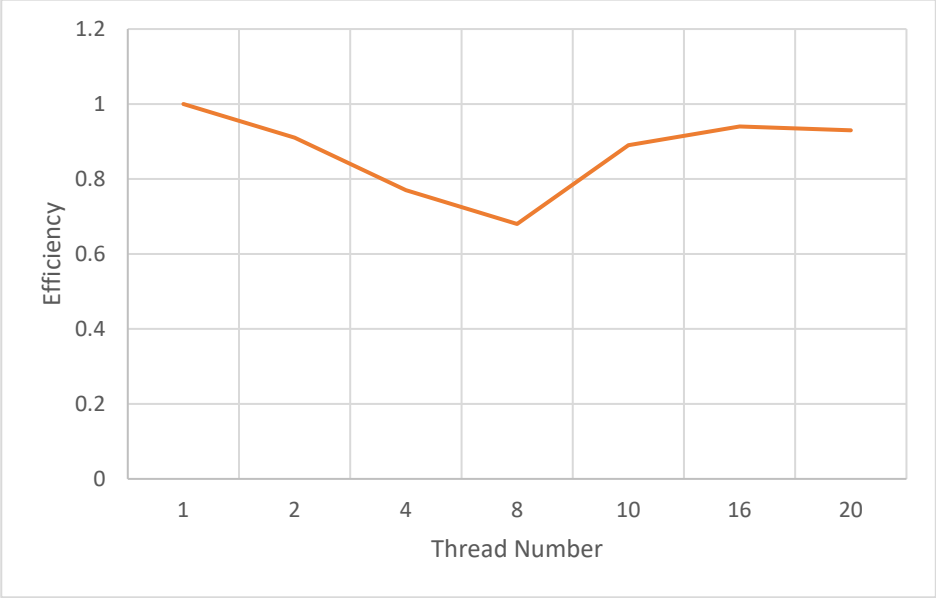
The results of part I are store in the /Homework#3/Part1 directory and the results of part II are store in the /Homework#3/Part2 in the GitHub repository[1].

The Table and plots below record the results of part II.

[1] https://github.com/yhuang86/UA-High_Performance_Computing

Test Case #	Problem Size	Max_Gen	Thread	Time(s)	Speed Up	Efficiency
1	5000x5000	5000	1	2024.53	1.0	1.0
2	5000x5000	5000	2	1115.37	1.81	0.91
3	5000x5000	5000	4	658.33	3.07	0.77
4	5000x5000	5000	8	374.33	5.41	0.68
5	5000x5000	5000	10	227.95	8.88	0.89
6	5000x5000	5000	16	135.11	14.98	0.94
7	5000x5000	5000	20	109.14	18.55	0.93





V. Analysis and Conclusion

The 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. When running code on the ASC cluster, the number of cores used for different test cases is the same as the thread number.

Different from the program in assignment#1, this program was modified from life.c in the blackboard as the solution of assignment#1. The reason for using life.c instead of the program in assignment#1 is that life.c is more sophisticated than the other program.

The program achieves the goal of implementing the game of life with the multithreaded method and improve the performance of the program compare with assignment#1. Each thread will spread the whole graph computation which can save more time than a single thread. After computing all cells in the next generation, a single thread will swap the pointers which are improved from letting master thread to swap pointer. To ensure the correctness of the program, the output files are compared with an example program from blackboard – life.c – with "diff" command.

From the speedup vs. thread number graph and timetable, it is clear that double the thread number, almost half the time taken which is meet the expectation that multithread can increase the performance of the program, and the trend speedup decrease when keeping increasing the number of threads.

However, the efficiency vs. thread number is a little counterintuitive which is the efficiency when thread number is 10, 16, and 20 are greater than those when thread number is 2, 4, and 8. The reason for this result is probably the ASC system delay and time for communication between the distributed nodes of the system.

The difficult part of this assignment is to make sure the flag records the change between two generations. To make sure the function of the flag variable, a local variable count was introduced to record the changes for each thread. After computation, the sum of the count will be assigned to the flag which indicates the number of cells changes between generations.

To further improve the program of the game of life, I reckon that paralleled more percentage of sequential program could be a good solution because more percentage sequential program be paralleled, more time saved for each iteration. Also, reduce the number of barriers in the parallel region can also save some time and increase performance.

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 hw#3.c is modified from life.c from the blackboard.