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

Homework # 4

By

Yichen Huang

CS 481 High-Performance Computing

October 28, 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 multiprocessor API – Message Passing Interface (MPI) will be employed to increase the performance of the program in assignment#1 with a multiprocessor's 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 processors(num_processors), 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 multiprocessors program, the number of processes will be read through the command line. The parallel section will start after all declared variables. The life and temp matrix will be allocated and initialize in the process 0. To evenly divided the life matrix by row, there is an array named count will calculate number of rows for each processor. Each processor will have own mylife and mytemp array with ghost cells surround to store the part of life and temp array. To scatter the life array to mylife array of each processor, MPI_Scatterv function has been applied. MPI_Sendrecv function has been applied to exchange the data in each process with the ghost cells from neighboring process.

After scatter the life array to mylife array and data exchange for each process, the program will let each process iterate the mylife array and calculate the status of next generation. The result will be stored into mytemp. A flag will record the number of changes between two generations and MPI_Allreduce will be used to calculate the total number of changes. If the flag is greater than 0, the mylife and mytemp pointer will be swap, the MPI_Sendrecv function will be applied to transfer row from mylife array to ghost cells of neighboring processes and start to calculate next generation. When the flag is 0 or hit the maximum generation, the loop will stop and gathering mylife to the life matrix in process 0 by MPI_Gatherv function.

At the end of the program, life and temp array will be manually released in the master processor. Each processor will also release the mylife and mytemp array manually. The result of life array will be print to the output file and store in the directory from the command line 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 multiprocessor version is correct comparing with the sequential version. Part one will check the output from the multithread version program and sequential program from blackboard with different size and max iteration. On local system, 4*4, 10*10, 1000*1000 size with different maximum iteration and processor number will be tested and compare with the output from life.c from blackboard. In supercomputer, 1000*1000 board size with different iteration number and processor number will be test. The diff command will be used to check the difference between the output file from sequential program and multiprocessors program.

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 processor number with 1, 2, 4, 8, 10, 16, and 20. With three runs, the average time taken will be recorded.

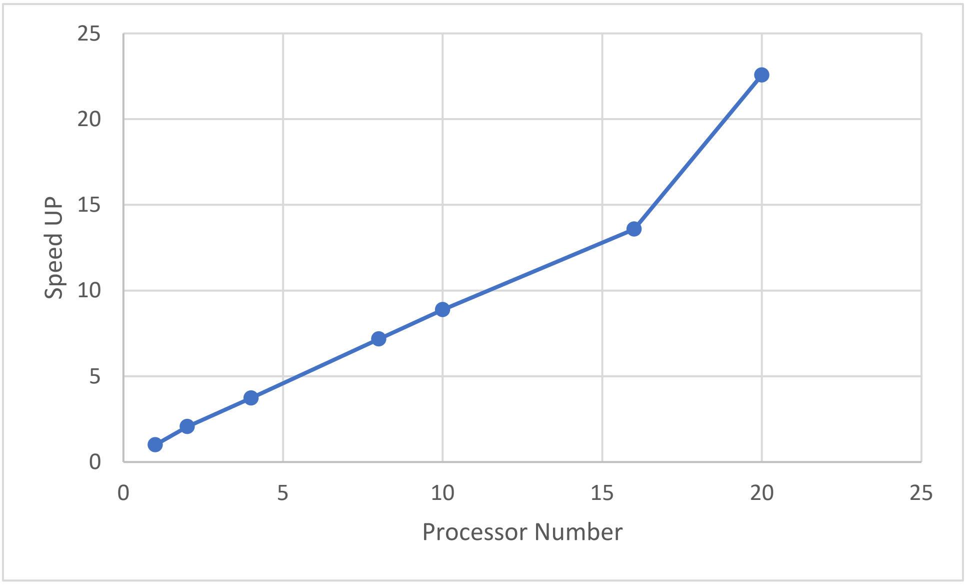
IV. Test Cases

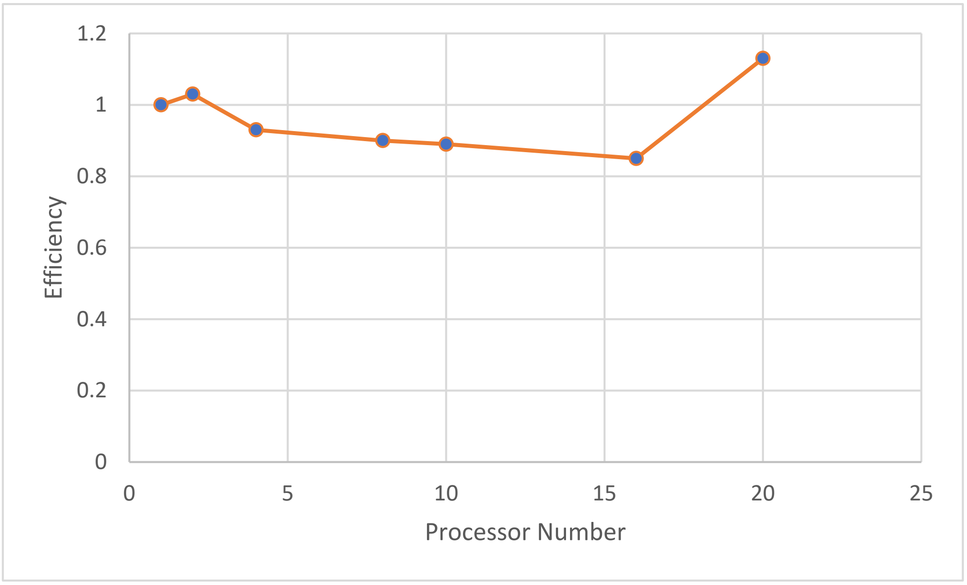
The results of part I are store in the /Homework#3/Part1 directory and the results of part II are store in the /Homework#3/Part2 in the GitHub repository[1].

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	Processor	Time(s)	Speed Up	Efficiency
1	5000x5000	5000	1	427.77	1.0	1.0
2	5000x5000	5000	2	207.27	2.06	1.03
3	5000x5000	5000	4	114.58	3.73	0.93
4	5000x5000	5000	8	59.7	7.17	0.90
5	5000x5000	5000	10	48.17	8.88	0.89
6	5000x5000	5000	16	31.49	13.58	0.85
7	5000x5000	5000	20	18.95	22.57	1.13





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 quad-core 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 number of processors described in Section III.

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 program in Assignment#1.

The program achieves the goal of implementing the game of life with the multiprocessor method and improve the performance of the program compare with assignment#1. The whole life array will be scattered evenly to each processor and compute next generation which can save more time than a single processor to do so. After computing all cells, the mytemp an mylife array will be swap and exchange the ghost cell with neighboring processors. 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. processor number plot, efficiency vs. processor number plot, and timetable, it is clear that double the thread number, almost half the time taken which is meet the expectation that multiprocessor programming can increase the performance of the program. The speed up between 4 and 16 processors have decrease a little which met the expectation that keep increasing processor number, the trend of speed up will decrease. From the efficiency with processors number plot, the efficiency between 4 processors and 16 processors are decreasing which is also met the expectation. However, when the processor number is 20, the speed up is greater than expected. The reason could be the program be assigned to a CPU have greater base frequency than other test cases. It is also possible that when running the 20 processors program, the frequency of memory is faster than other test case which reduce the running time of the program.

Comparing with the result from assignment#3, which is multithread version (OpenMP) of game of life, the overall running time of multiprocessor version is slower than the multithread version program. The speed up and efficiency also shows that the speed up and efficiency of multiprocessor version is slightly lower than multithread version. The reason for the difference between two version is that in multiprocessor program, each processor has own part of memory which is different than the multithread version. The communication between processors will consume lots of time for waiting and receiving. All thread in OpenMP shares memory, they won't speed time on communication than multiprocessor version.

To further improve and optimize the program, I think I use lots of MPI_Barrier function in the program. The barrier will make sure all process can go to next step when they all call the function which can make sure the stable of the code. However, the abuse the function will let all processors waste lots of time during waiting.

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.