Game of Life MPI Simulation

By
Md Mashiur Rahman Chowdhury
CS 732 Parallel Computing
April 3, 2018

Problem Specification

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a board game with two-dimensional array of cells. Each cell represent the state of an organism and has eight neighboring cells (left, right, top, bottom, top-left, bottom-right, top-right, bottom-left). It is assumed that, array wraps around in both dimensions which means that the left boundary is adjacent to the right boundary and the top boundary is adjacent to the bottom boundary (often referred to as periodic boundary conditions). Each cell can be in two states : alive or dead. The game starts with an initial state (alive or die) and multiply in the next iteration according the following rules :

1. If a cell is “alive” in the current generation, then depending on its neighbor’s state, in the next generation the cell will either live or die based on the following conditions:
    - Each cell with one or no neighbor dies, as if by loneliness.
    - Each cell with four or more neighbors dies, as if by overpopulation.
    - Each cell with two or three neighbors survives.
2. If a cell is “dead” in the current generation, then if there are exactly three neighbors "alive" then it will change to the "alive" state in the next generation, as if the neighboring cells gave birth to a new organism.

The above rules apply at each iteration (generation) so that the cells evolve, or change state from generation to generation. Also, all cells are affected simultaneously in a generation (i.e., for each cell you need to use the value of the neighbors in the current iteration to compute the values for the next generation).

In this assignment we have implemented the MPI version of the above problem. We have divided the board among the process, done computation on each of it, shared the state after each iteration and finally computed result. We have followed above steps to get maximize speed up, efficiency. After implementing and testing the result we have run the experiment in the dmc cluster.

Program Design

We have used a 2D array of size (N+2) * (N+2) to keep the state of the board. To implement the iteration we need to define another 2D array of same dimension to keep the mirror result. Though N * N array is enough to keep the board state but we have decided to keep the ghost cell for ease of calculation. We have initialized those ghost cells with the corresponding values from opposite boundary. As we have

these ghost cells it was easy to calculate alive cells and make it simpler to check the cells along the boundary.
In the MPI implementation part, we have decided to parallelize the program row wise. That means if there are n rows and p process then each process would compute the n/p rows of board. Moreover we need to share the neighbouring rows with the sibling process. Suppose for process n we have shared the last row, first row with process n+1 and n-1 respectively. The last row of n is the top neighbour of process n+1's first row and at the same time first row of process n is the bottom row neighbour of process n-1's last row.

Testing Plan

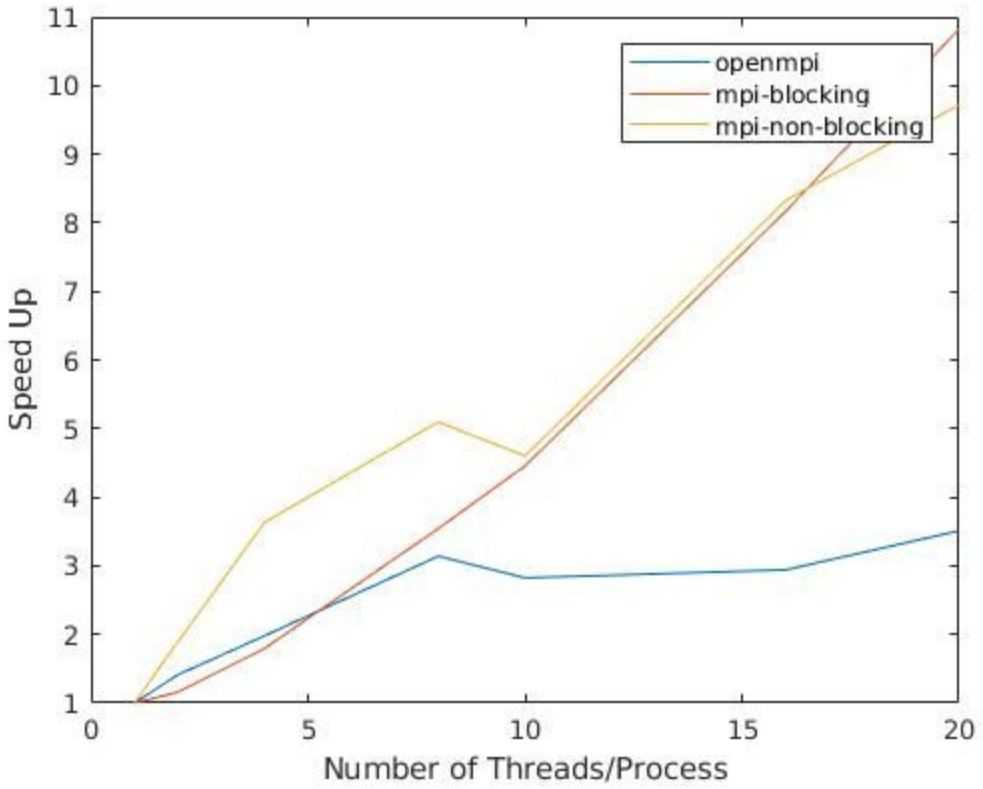
We have done the experiment for number of process 1, 2, 4, 8, 10, 16, 20 for the board size 5000 * 5000. While running experiments in the cluster we have assigned appropriate number of cores according to process to get maximum speed up and efficiency. Suppose if in the MPI program we are running with n process we have assigned n cores to get max speed up. We have run each experiment for 3 times and then calculated average. We have done 5000 iteration for each of the segment of the board without considering whether the board state changed or not. After finishing the iterations in each of the process, the process send the master a flag indicating that it is finished. Master waits until it gets all of the flags from each of the process and then terminates. We have implemented both the blocking and non-blocking versions of mpi and populated the results in next sections.

Test Case

system	1	2	4	8	10	16	20
MPI(Blocking)	44.059 minutes	38.044 minutes	24.55 minutes	12.44 minutes	9.888 minutes	5.3979 minutes	4.070 minutes
Speed Up	1	1.158	1.795	3.5417	4.4558	8.1622	10.825
Efficiency	1	0.579	0.44875	0.447125	0.44558	0.5101375	0.54125
MPI(Non Blocking)	51.24759 minutes	27.1269 minutes	14.1038 minutes	10.055 minutes	11.1366 minutes	6.158 minutes	5.273 minutes
Speed Up	1	1.889	3.6336	5.0967	4.602	8.322	9.719
Efficiency	1	0.9446	0.908	0.637	0.4602	0.520	0.4859
OpenMp	65.15 minutes	46.20 minutes	32.88 minutes	20.77 minutes	23.07 minutes	22.18 minutes	18.57 minutes
Speed Up	1	1.41	1.98	3.14	2.824	2.94	3.51
Efficiency	1	0.705	0.495	0.3925	0.2824	0.18375	0.1755

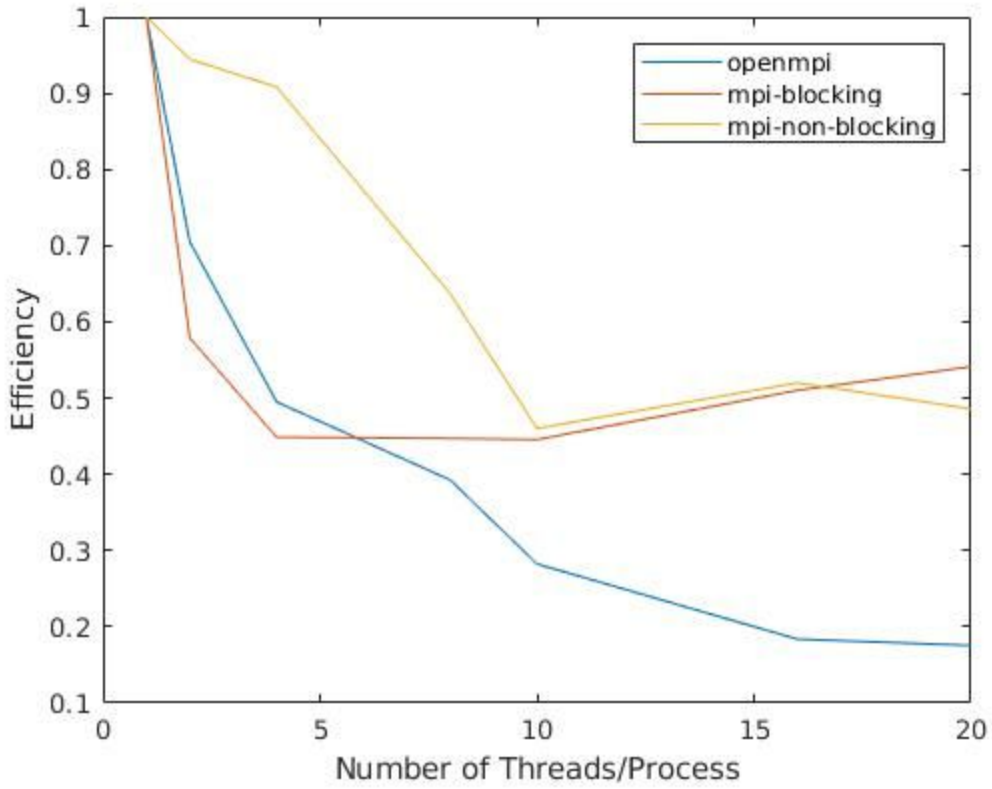
Analysis & Conclusion

Speed Up :



In the above graph we can see the speed up result of openmp, mpi blocking and mpi-non-blocking program. We have run this programs on process 1, 2, 4, 8, 10, 16, 20 for the same input size (5000 * 5000). From the graph we can see that, the blocking version of mpi code speed up quite linearly with respect to the other 2 versions. Here we can see that clearly the mpi versions are much faster that the openmp version. We can see some skewed result for the non-blocking version though there is not that much difference in those 2 programs other than the mpi library. There might be many reasons to get this kind of results. Suppose when we were running the 2 programs on different time the dmc clusters load might be different. There might be some mpi implementation issue here also.

Efficiency :



In the above graph we can see the efficiency result of openmp, mpi blocking and mpi-non-blocking program. We have run this programs on process 1, 2, 4, 8, 10, 16, 20 for the same input size (5000 * 5000). From the graph it is clear that the mpi versions are more efficient with respect to the openmp version as the number of process grows.

Reference

1. Problem Specification got from the first Assignment
2. https://computing.llnl.gov/tutorials/openMP/.
3. http://www.cas.mcmaster.ca/~nedialk/COURSES/mpi/Lectures/lec2_1.pdf

Gitlab Information:

Project :
https://gitlab.cs.uab.edu/mashiur/cs732/tree/master/Assignment-4

Point to Point Blocking :
https://gitlab.cs.uab.edu/mashiur/cs732/blob/master/Assignment-4/main_mpi.cpp
Experiment Result :
https://gitlab.cs.uab.edu/mashiur/cs732/tree/master/Assignment-4/result

Point to Point Non Blocking :
https://gitlab.cs.uab.edu/mashiur/cs732/blob/master/Assignment-4/non-blocking/main_mpi.cpp
Experiment Result :
https://gitlab.cs.uab.edu/mashiur/cs732/tree/master/Assignment-4/non-blocking/result