CONWAY'S "GAME OF LIFE"

Homework # 1

By

Marcelo Torres

CS 481/582 High Performance Computing
September 12, 2024

Problem Specification:

The objective of this assignment is to design and implement a simulation of Conway's "Game of Life," a cellular automaton devised by mathematician John Horton Conway. The Game of Life is played on a two-dimensional grid of cells, where each cell can be either alive or dead. The state of the cells evolves over discrete time steps according to a set of rules based on the states of neighboring cells. The rules are applied differently to living and dead cells. For living cells, if a cell has less than two living neighbors, it dies by underpopulation. If a cell has two or three live neighbors, it remains alive. Finally, if a live cell has more than three neighbors, it dies by overpopulation. Dead cells, on the other hand, remain dead unless they have exactly three live neighbors, in which case they become live cells by reproduction.

The game of life has simple rules that, when extrapolated to large boards or patterns, can be used to create complex patterns and automatons. There are many resources online demonstrating complex self-replicating designs in Conway's game of life. The game is a great demonstration of an idea often present in CS algorithms: a small set of rules can create extremely complex and unpredictable behavior. In many cases, the most complex algorithms are based on a small, well curated set of rules derived from the behavior of many complex solutions to a problem. However, for this project, no complex analysis was needed.

The goal of this project is to demonstrate the computing requirements of Conway's "Game of Life" by writing and running the program with larger and larger iteration counts and field sizes. To that end, the program designed must use a dynamically allocated game board and accept both the size and iteration count at runtime.

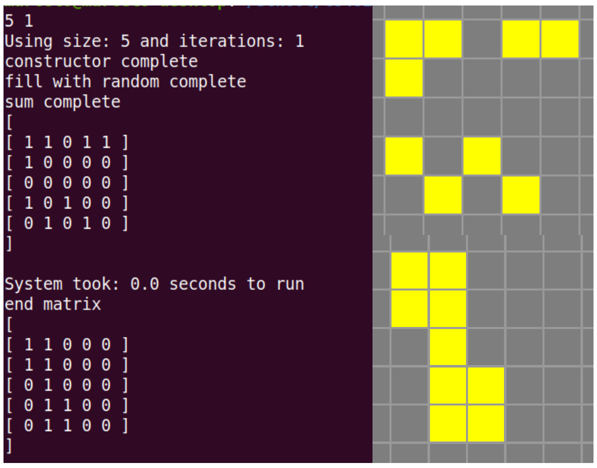
Program Design:

My implementation of the game board has changed throughout my testing. Originally, I designed a game board using a vector of 64-bit unsigned integers. I then stored one cell in each byte. Bitwise operations were then used to read or update a cell. Further development lead to the introduction of an "offset" value that allowed me to store multiple copies of the grid side by side in the same vector. After testing, however, this system proved to be too slow. At first, I suspected that the vector object was slowing down the program because of concurrent access, so I created a game board based on an array of unsigned 64-bit integers with the same approach. However, that did not yield a performance increase. I believe that the overhead introduced by the calculations necessary to access each element ended up overcoming any computing advantages gained from bitwise operations and efficient storage. Finally, I settled on a 2D array of unsigned 8-bit integers. This structure yielded a significant performance increase over the other two and is used in the final version.

The update method is based on a threadpool and splits the game board into horizontal rectangular sections equal to the number of threads used. Because the board is split into multiple sections and offset into multiple boards, no critical sections exist. One board is only read, and the other board is only written to once in each location for each update. The update method repeatedly calculates the number of living neighbors for each cell and updates that cell accordingly. Additionally, the update method tracks whether an update has occurred and returns false if an entire generation passes without updating a single cell.

Testing Plan

The game logic was tested using small board sizes of 5x5 and a single iteration. The results were printed and compared to the website https://playgameoflife.com/ which provides a free interactive implementation of Conway's "Game of Life." An example is shown below.



```
5 1
Using size: 5 and iterations: 1
constructor complete
fill with random complete
sum complete
[
[ 1 1 0 1 1 ]
[ 1 0 0 0 0 ]
[ 0 0 0 0 0 ]
[ 1 0 1 0 0 ]
[ 0 1 0 1 0 ]
]

System took: 0.0 seconds to run
end matrix
[
[ 1 1 0 0 0 ]
[ 1 1 0 0 0 ]
[ 0 1 0 0 0 ]
[ 0 1 1 0 0 ]
[ 0 1 1 0 0 ]
]
```

Figure 1: A validation of the game logic

The above figure depicts a printout of the game state before and after an iteration, as well as a graphic representation of the state taken from the website listed above.

Populating the board is accomplished using a uniform distribution, which yields close to 50% population in most tests. Multiple tests are computed to account for variability. The results are as follows.

Test Case #	Problem Size	Max. Generations	Time Taken (s)
1	1000x1000	1000	5.495667
2	1000x1000	5000	28.553
3	5000x5000	1000	132.419
4	5000x5000	5000	661.316
5	10000x10000	1000	466.5505
6	10000x10000	5000	2359.622

Table 1: Test Results

The tests were all run using a release version of the code with the –02 compiler flag.

Test Cases

All Tests were run using a random, uniform distribution to populate the board. In measurements on boards of 1000x1000 cells, the distribution produced board that were 50% living cells with a tolerance of ±0.05%. All tests were run using 10 threads on an Ubuntu 20.04 desktop with 32Gb of ram and an AMD Ryzen 7 3700X 8-Core Processor. They were compiled using CMake and GCC version 4.9.3.

Analysis and Conclusions

The results of the project were mostly as expected. Increased game board size and iteration count lead to proportionate increases in run time. These results demonstrate that the expected number of computations is a good indicator of the expected run time of a program. The expected behavior of computation count would be to grow exponentially with the size of the game board and linearly with the time number of iterations. However, there is a discrepancy in this growth demonstrated at the jump from 5000x5000 elements to 10000x10000 elements. At that jump, the execution time decreases significantly. I suspect that this is due to the multithreaded nature of the program. This indicates that more overhead is introduced by successive iterations than by increasing the game board size. These results suggest that the overhead introduced by setting up multithreading and the performance increased introduced by multithreading the computation of a new game state outweigh the exponential growth in size of the game board. In future iterations of the program, I would investigate methods to reduce the overhead in each successive iteration by precomputing the sections of the board used by the multithreaded process.

My goal when beginning the project was to be able to competently run all game board sizes and iterations. Unfortunately, I do not believe that I have achieved that to the extent which I desired. The highest sizing I believe I was successful on was the 5000x5000 size with 1000 iterations. After that, runtimes become too high. In the future, I would make multiple changes to improve performance. First, I would store the game board as a continuous row of unsigned 8-bit integers to hopefully improve cache performance. Second, I would track which cells could possibly be updated and which cells could not, a system that should significantly reduce computations even on an evenly populated board. Lastly, I would change the method of calculating neighbors for each cell to take advantage of the sequential nature of that process. A simple way to do this would be to use a window approach where three vertical cells are read at once and then stored and used over the course of the next three calculations. I believe that these changes could significantly improve the performance of the algorithm.