MULTITHREADED HIGH-PERFORMACE OF CONWAY'S GAME OF LIFE,

Homework 3

By

Gowtham Prasad

CS 581 High Performance Computing
October 16. 2024

Problem Specification:

The objective of this project is to design and implement an efficient multithreaded version of the "Game of Life" program using OpenMP.

The goal of the C code is to create and run a simulation of Conway's Game of Life, a cellular automaton in which a two-dimensional array of cells evolves according to predefined rules. It was American mathematician John Von Neumann who first proposed the concept of a universal constructor, or a software that could replicate itself automatically by analyzing data, back in the 1940s, when life first originated. Von Neumann created a discrete game with a maximum of 29 distinct states for each cell in a two-dimensional grid of square cells to explore this problem. In this game, a cell's state in each generation is determined by the state of adjacent cells. This universal constructor fascinated British mathematician John Conway, who used it as motivation to study a new computable discrete world. Alan Turing's idea of a “universal computer,” or a device that can mimic any computational process by adhering to a brief set of instructions, served as the foundation for this study. Conway thus began to search for a computable “simple universe,” a condensed set of states and laws. After experimenting with numerous configurations on a two-dimensional grid to establish an optimal balance between extinction and unlimited cellular proliferation, he named this set of rules the Game of Life.

Each cell is categorized as either “alive” or “dead” based on the influences of its eight neighbors. The simulation must precisely update the grid over a number of generations and terminate when either a stable state is reached, or the maximum number of generations is reached. Only if a surviving cell in this generation has two or three neighbors will it survive until the next generation; otherwise, it dies from loneliness or overcrowding. Even a dead cell can spontaneously regenerate into a living one in the next generation if it has exactly three neighbors in the present generation. The seed of the system is the original pattern and is an N x N matrix. The criteria are applied simultaneously to all of the seed's cells, whether they are living or dead, to generate the first generation. Births and deaths take place at the same time, and this discrete moment is frequently referred to as a tick. Every generation is solely dependent on the one before it. The guidelines are consistently followed in order to produce new generations.

Program Design:

The implementation of the Game of Life uses OpenMP for parallelization, aiming to maximize computational efficiency by dividing the grid among threads. The board is initialized as an N x N matrix with random values, where cells can either be alive or dead. The edges of the matrix are set to 0 to represent dead cells, creating a boundary that simplifies calculations. The core of the program leverages OpenMP to split the grid rows across threads. Using the #pragma omp parallel directive, each thread is assigned a subset of the grid to process independently. Thread IDs (tid) are used to calculate the range of rows that each thread will work on. The number of threads (numThreads) is passed as a parameter to the program, allowing flexible

testing of parallel performance. For each cell in the grid, the number of alive neighbors is computed by summing the values of the eight neighboring cells. Based on these values, cells are updated according to Conway's rules: a live cell with fewer than two or more than three neighbors die, while a dead cell with exactly three neighbors becomes alive. After each thread completes its portion of the grid, the #pragma omp barrier ensures synchronization, preventing race conditions when threads swap the board and newBoard arrays for the next generation. The grid size is divided evenly among the threads, with the last thread handling any extra rows if the grid size isn't perfectly divisible by the number of threads. This minimizes idle time among threads and balances the workload efficiently.

Testing Plan:

The testing is divided into two main categories: correctness and performance. In terms of correctness, start with a small, predefined grid (e.g., 5x5) and verify the outputs match expected results for known patterns like still lifes and oscillators. The generated output files from the program are manually inspected to ensure that the evolution of the grid follows Conway's rules. Additionally, larger random grids (e.g., 500x500) are initialized and checked for stability or termination after a set number of generations. The output from different thread configurations is compared to ensure consistent behavior. In terms of performance, vary the number of threads from 1 to 20 to measure how well the program scales. Execution times are collected for each thread count, with both Intel and GNU compilers. These results are recorded and plotted to observe speedup and efficiency. Test the program with different grid sizes (e.g., 100x100, 5000x5000) to evaluate the scalability of the solution as the problem size increases. Execution time, speedup, and efficiency are measured for each grid size. Furthermore, test that cells on the grid edges correctly wrap around due to periodic boundary conditions. This ensures the program handles the edges as expected and avoids artifacts from boundaries.

Test Cases:

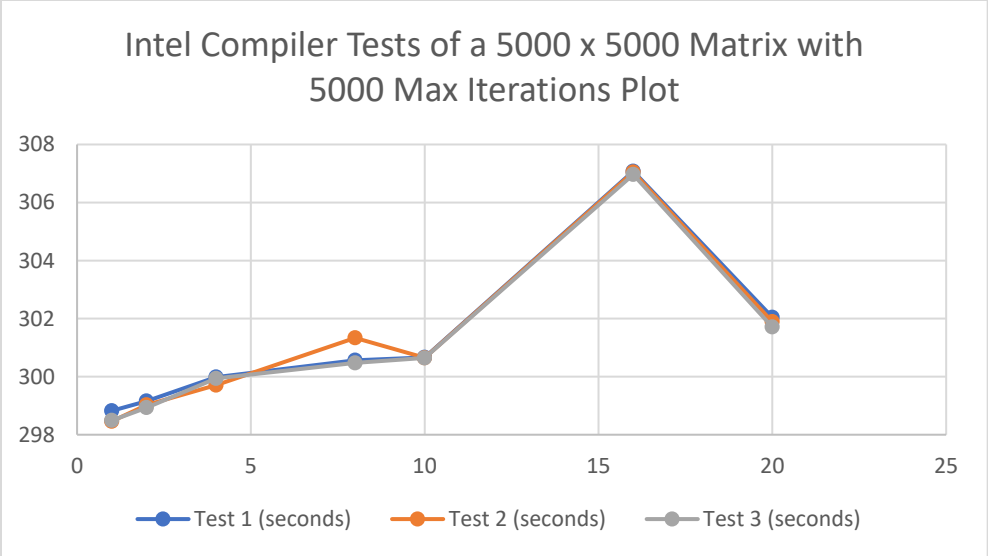
Tests 1 through 3 for all compilers had to following conditions in the HPC:

- Class queue
- A time limit of 12:00:00 HH:MM:SS
- A memory limit of 8gb
- 1 core

Thread Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time of All Test Cases (seconds)
1	298.8247454	298.461361	298.491011	298.5923725
2	299.160284	299.027499	298.934138	299.0406403
4	299.987534	299.708038	299.940509	299.8786937
8	300.563874	301.335809	300.473336	300.7910063
10	300.663345	300.649781	300.643745	300.6522903

16	307.078806	307.04697	306.96383	307.0298687
20	302.045026	301.894783	301.711853	301.8838873

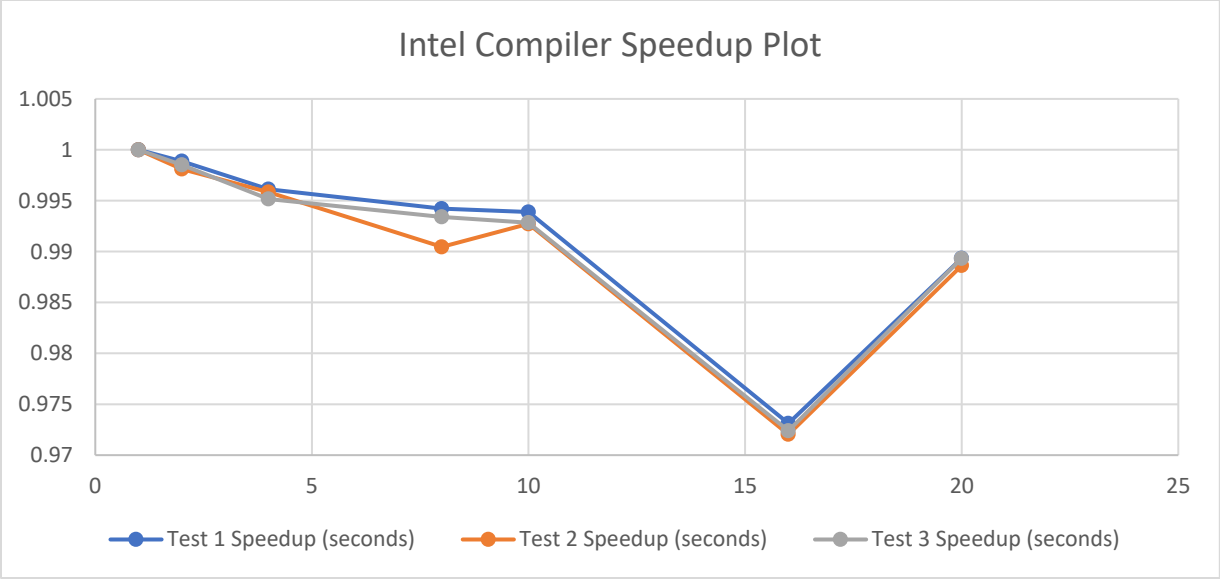
Table 1. Intel Compiler Tests of a 5000 x 5000 Matrix with 5000 Max Iterations



Graph 1. Intel Compiler Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Plot

Thread Count	Test 1 Speedup (seconds)	Test 2 Speedup (seconds)	Test 3 Speedup (seconds)
1	1	1	1
2	0.998878398	0.998106736	0.998517643
4	0.996123877	0.995840362	0.995167382
8	0.99421378	0.990460981	0.993402659
10	0.993884856	0.992721032	0.992839585
16	0.973120709	0.972038125	0.972397989
20	0.989338409	0.988627091	0.989324775

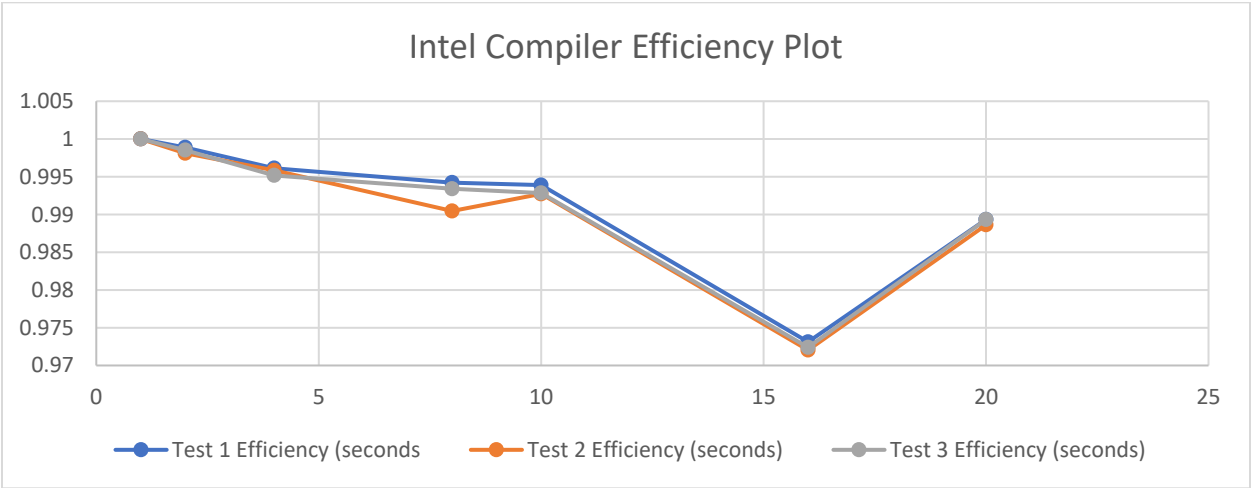
Table 2. Intel Compiler Speedup Table



Graph 2. Intel Compiler Speedup Plot

Thread Count	Test 1 Efficiency (seconds	Test 2 Efficiency (seconds)	Test 3 Efficiency (seconds)
1	1	1	1
2	0.998878398	0.998106736	0.998517643
4	0.996123877	0.995840362	0.995167382
8	0.99421378	0.990460981	0.993402659
10	0.993884856	0.992721032	0.992839585
16	0.973120709	0.972038125	0.972397989
20	0.989338409	0.988627091	0.989324775

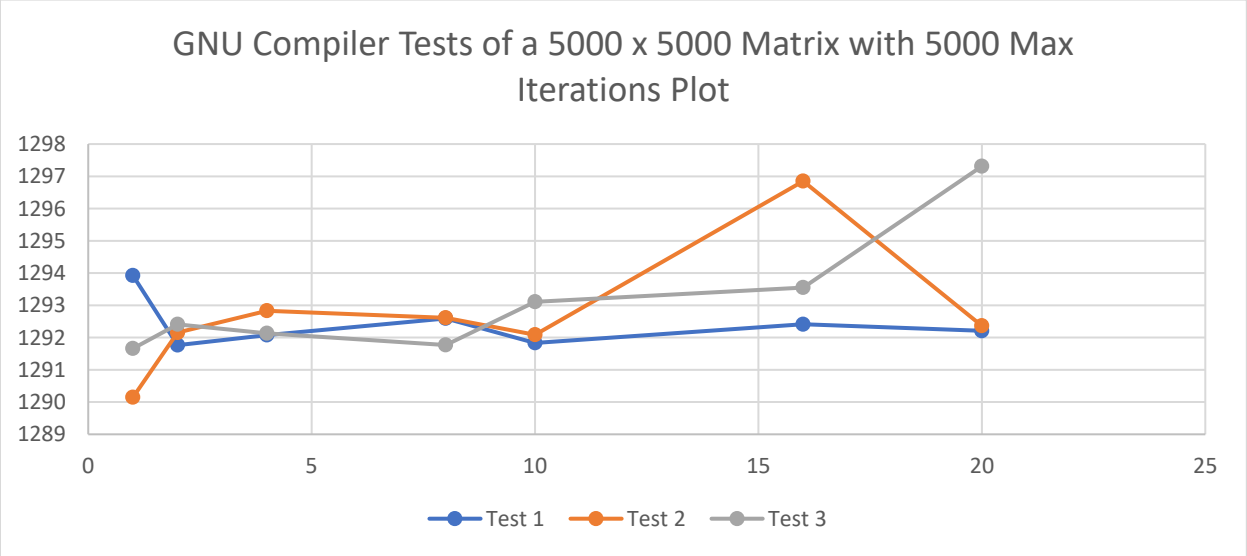
Table 3. Intel Compiler Efficiency Table



Graph 3. Intel Compiler Efficiency Plot

Thread Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time of All Test Cases (seconds)
1	1293.923993	1290.146806	1291.659701	1291.910167
2	1291.764791	1292.15419	1292.409815	1292.109599
4	1292.077669	1292.832278	1292.131933	1292.347293
8	1292.594139	1292.614459	1291.771002	1292.326533
10	1291.835429	1292.088154	1293.109431	1292.344338
16	1292.411185	1296.849443	1293.551098	1294.270575
20	1292.20873	1292.364649	1297.311543	1293.961641

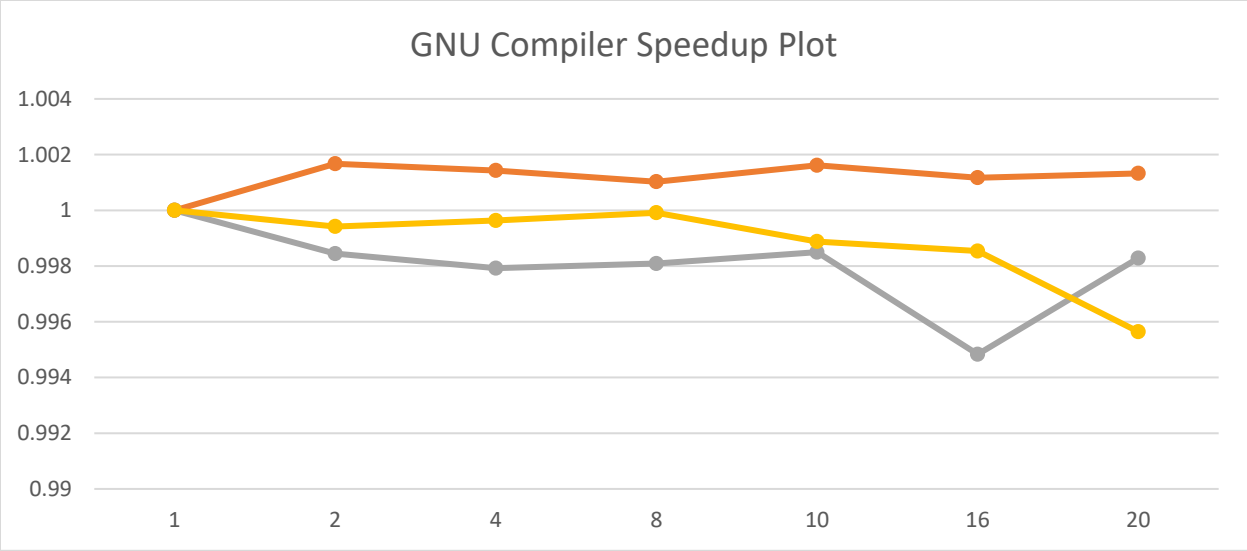
Table 4. GNU Compiler Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Table



Graph 4. GNU Compiler Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Plot

Thread Count	Test 1 Speedup (seconds)	Test 2 Speedup (seconds)	Test 3 Speedup (seconds)
1	1	1	1
2	1.001671513	0.998446483	0.999419601
4	1.001428957	0.997922799	0.999634533
8	1.001028826	0.99809096	0.999913838
10	1.001616742	0.998497511	0.998878881
16	1.001170531	0.9948316	0.998537826
20	1.001327388	0.998283888	0.99564342

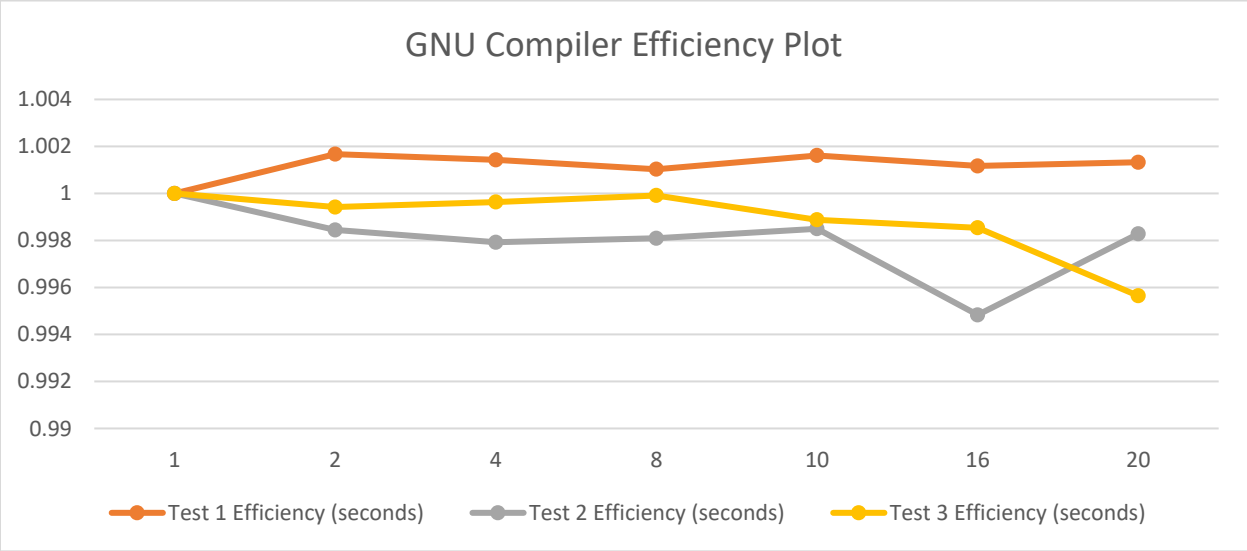
Table 5. GNU Compiler Speedup Table



Graph 5. GNU Compiler Speedup Plot

Thread Count	Test 1 Efficiency (seconds)	Test 2 Efficiency (seconds)	Test 3 Efficiency (seconds)
1	1	1	1
2	1.001671513	0.998446483	0.999419601
4	1.001428957	0.997922799	0.999634533
8	1.001028826	0.99809096	0.999913838
10	1.001616742	0.998497511	0.998878881
16	1.001170531	0.9948316	0.998537826
20	1.001327388	0.998283888	0.99564342

Table 6. GNU Compiler Efficiency Table



Graph 6. GNU Compiler Efficiency Plot

Analysis and Conclusions:

The performance testing revealed key insights into how the multithreaded implementation of Conway's Game of Life scales with different numbers of threads. The Intel compiler demonstrated that increasing the number of threads beyond 8 does not yield significant speedup, with performance plateauing around 10 threads. The efficiency, particularly for higher thread counts (16 and 20), decreases as synchronization overhead and memory bandwidth contention become limiting factors. The GNU compiler showed higher overall execution times compared to Intel, with minimal speedup as the number of threads increased. This could be attributed to less optimized handling of parallelism by the GNU compiler. Even at 20 threads, the program's speedup remained close to 1, indicating poor scalability under this setup. The primary limitation on scalability appears to be memory bandwidth. With each thread attempting to access neighboring cells in the shared grid, cache misses increase, resulting in slower performance as thread counts increase. The barrier synchronization between generations introduces overhead, particularly as the number of threads increases. This is evident in the decreasing efficiency at higher thread counts. While the multithreaded implementation achieves modest speedup for smaller numbers of threads, it faces diminishing returns beyond 8 threads. Optimizing memory access patterns or exploring alternative parallelization strategies (such as dynamic scheduling) could improve performance. Additionally, storing the grid in a 1D array rather than a 2D array would reduce the overhead of memory access, as multidimensional arrays in C can cause cache misses due to non-contiguous memory locations, and instead of applying #pragma omp barrier after every generation, I could use double-buffering or asynchronous update techniques to reduce the number of synchronization points. Let each thread work independently on its portion of the grid and only synchronize at key points (like at the end of a set of generations). Overall, the Intel compiler outperformed the GNU compiler, likely due to more efficient multithreading optimizations.

References:

1. Code sample matrixPassingValues.c, Md Kamal Chowdhury, September 12, 2024
2. Code sample life.c, Md Kamal Chowdhury, September 24, 2024
3. Code sample matmul_2d_parallel_regionV1.2.c, Md Kamal Chowdhury, September 24, 2024
4. "Play John Conway's Game of Life." Playgameoflife.com, playgameoflife.com/. Accessed September 10, 2024

Github Link: https://github.com/gdprasad1201/CS-581-HPC-Fall-2024