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

Homework 3

By

Gowtham Prasad

CS 581 High Performance Computing
October 17. 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:

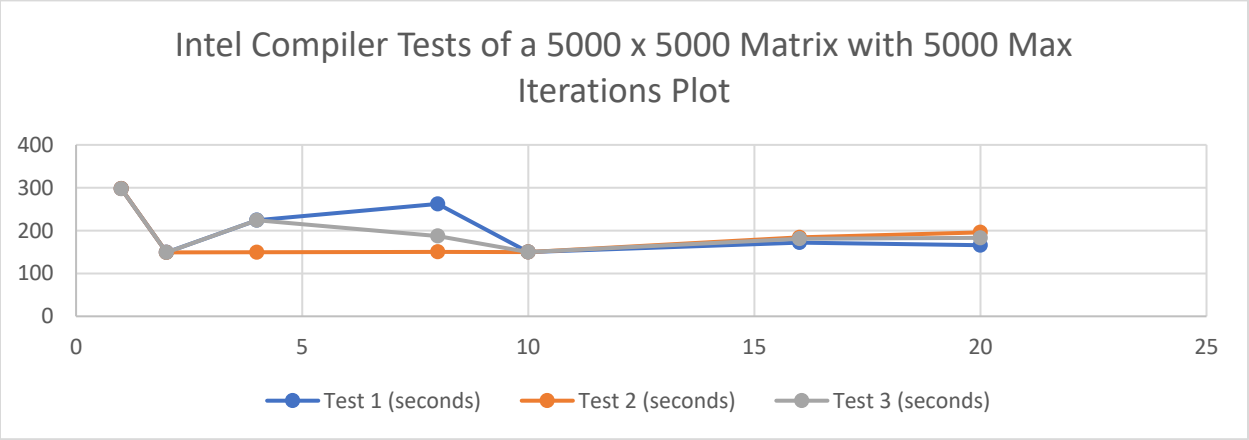
Tests 1 through 3 for all compilers experienced the following conditions in the HPC:

- Sent to the class queue
- A time limit of 12:00:00
- A memory limit of 8gb
- 2 cores

Thread Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
1	297.76413	297.820294	297.523469	297.702631
2	149.164036	149.135781	149.030508	149.1101083
4	224.310129	149.503696	224.109767	199.307864
8	262.23937	150.426794	187.255699	199.9739543
10	150.060349	150.079137	150.056268	150.0652513

16	172.101765	184.125904	181.025021	179.08423
20	165.761088	195.860912	183.015534	181.5458447

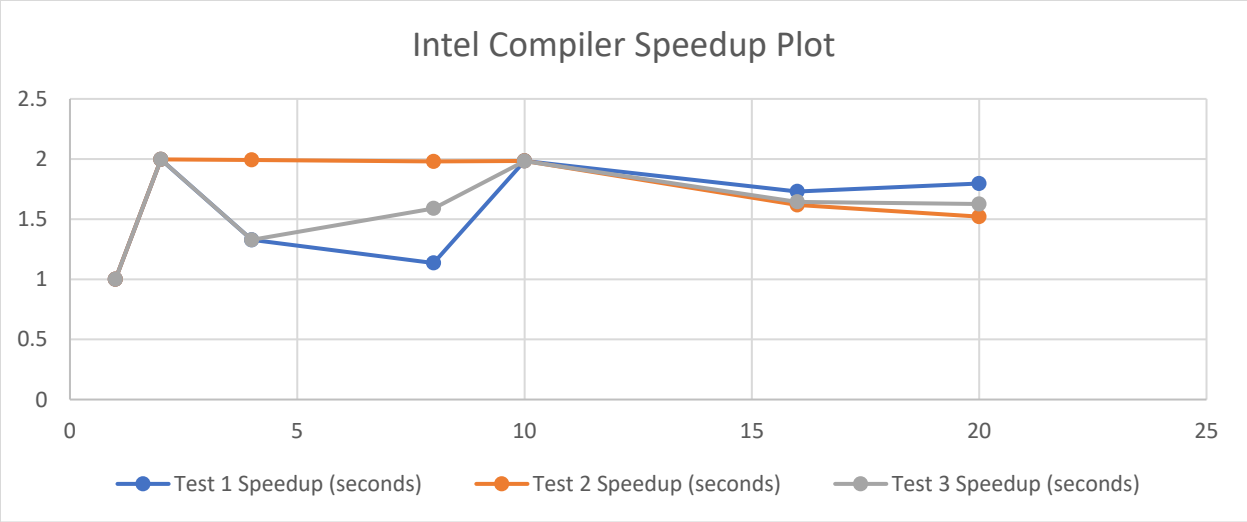
Table 1. Intel Compiler Tests of a 5000 x 5000 Matrix with 5000 Max Iterations



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	1.996219317	1.996974113	1.996393041
4	1.327466269	1.992059741	1.327579217
8	1.135466921	1.979835414	1.588862024
10	1.984295865	1.984421685	1.982746026
16	1.730163139	1.617481775	1.643548872
20	1.796345171	1.520570342	1.62567331

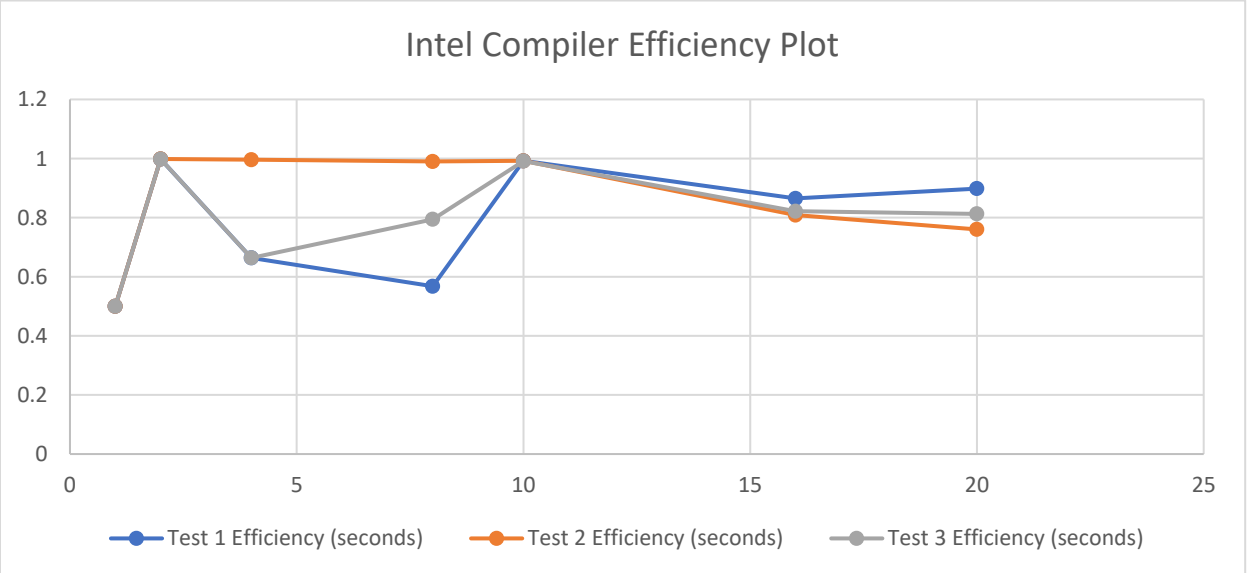
Table 2. Intel Compiler Speedup Table



Graph 2. Intel Compiler Speedup Plot

Thread Count	Test 1 Efficiency (seconds)	Test 2 Efficiency (seconds)	Test 3 Efficiency (seconds)
1	0.5	0.5	0.5
2	0.998109658	0.998487057	0.99819652
4	0.663733134	0.996029871	0.663789609
8	0.56773346	0.989917707	0.794431012
10	0.992147932	0.992210843	0.991373013
16	0.86508157	0.808740887	0.821774436
20	0.898172586	0.760285171	0.812836655

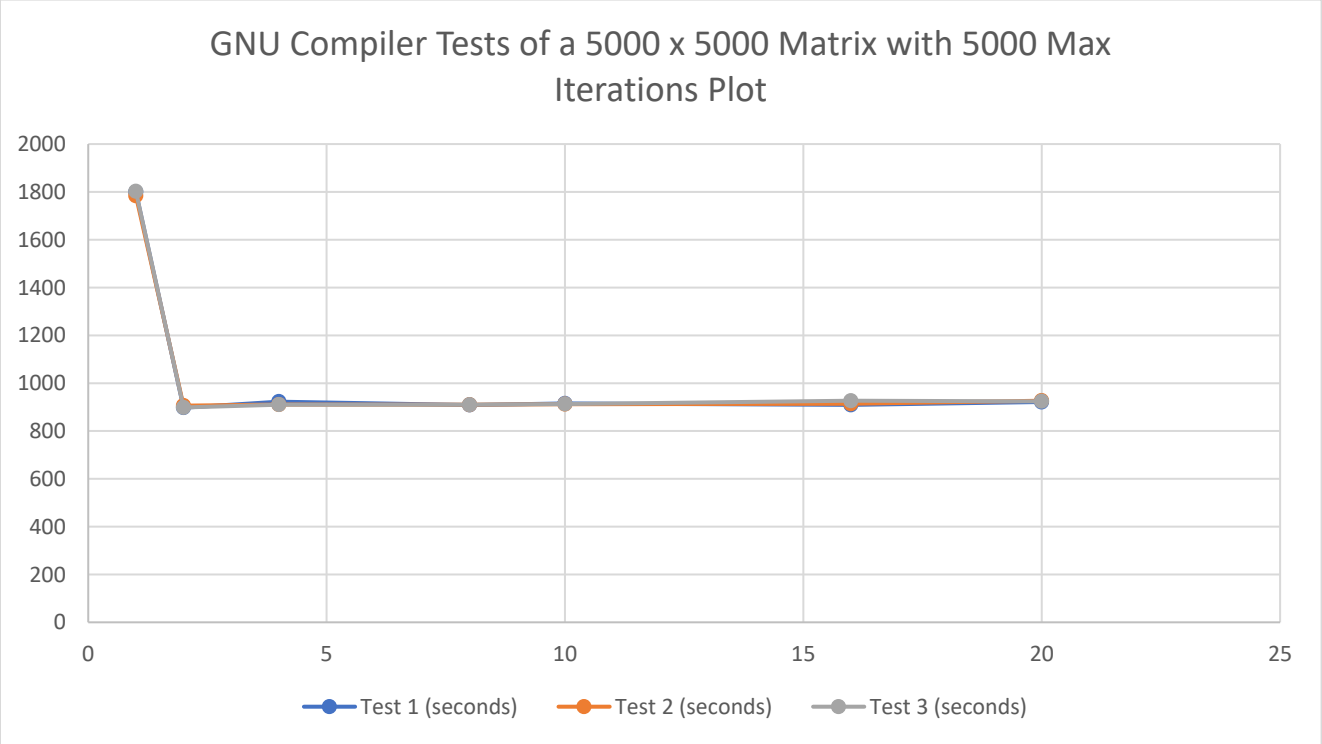
Table 3. Intel Compiler Efficiency Table



Graph 3. Intel Compiler Efficiency Plot

Thread Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
1	1799.364529	1783.781085	1802.136002	1795.093872
2	897.44526	906.074143	898.483474	900.6676257
4	923.043341	910.352948	910.09084	914.4957097
8	908.972273	909.825124	909.815106	909.537501
10	915.150066	912.038139	913.001354	913.3965197
16	909.326932	913.80911	926.679901	916.6053143
20	921.324233	926.780948	924.126947	924.077376

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

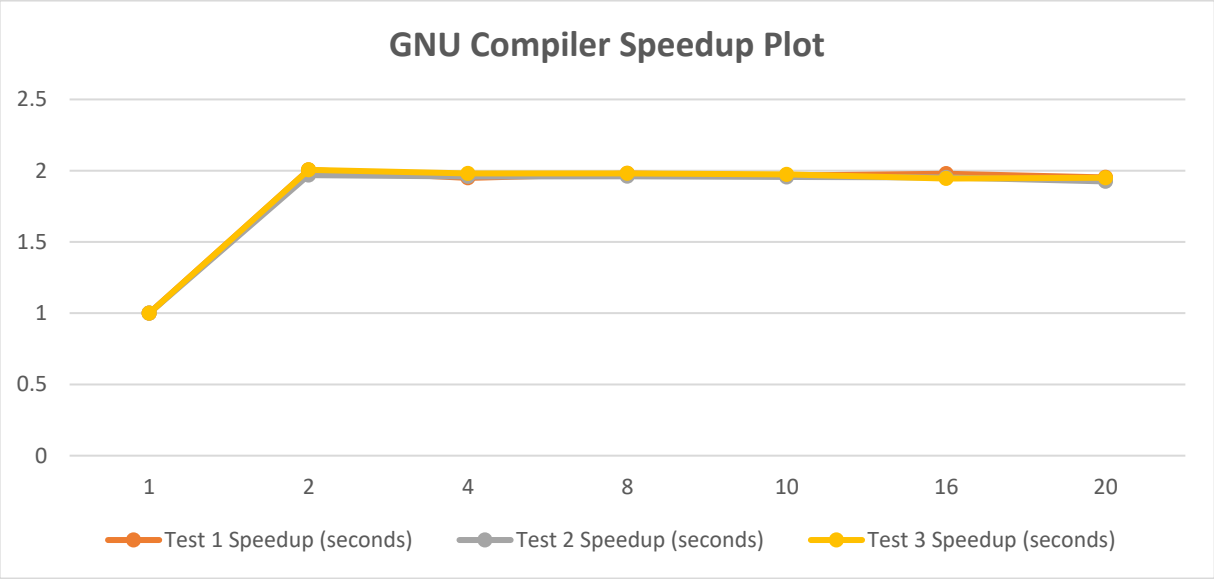


Graph 4. GNU Compiler Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Plot

Thread Count	Test 1 Speedup (seconds)	Test 2 Speedup (seconds)	Test 3 Speedup (seconds)
1	1	1	1
2	2.004985272	1.968692186	2.005753087
4	1.949382493	1.959439016	1.980171564
8	1.979559314	1.960575761	1.980771687
10	1.966196142	1.955818522	1.973859068
16	1.978787239	1.952028127	1.944723307

20	1.953019865	1.924706252	1.950095718

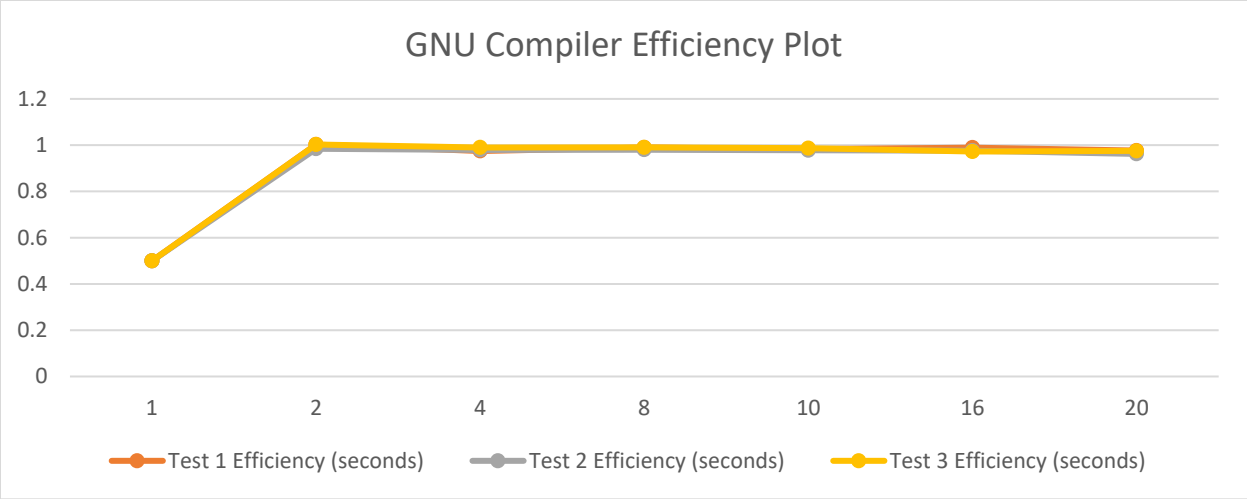
Table 5. GNU Compiler Speedup Table



Graph 5. GNU Compiler Speedup Plot

Thread Count	Test 1 Efficiency (seconds)	Test 2 Efficiency (seconds)	Test 3 Efficiency (seconds)
1	0.5	0.5	0.5
2	1.002492636	0.984346093	1.002876544
4	0.974691246	0.979719508	0.990085782
8	0.989779657	0.980287881	0.990385843
10	0.983098071	0.977909261	0.986929534
16	0.989393619	0.976014063	0.972361654
20	0.976509932	0.962353126	0.975047859

Table 6. GNU Compiler Efficiency Table



Graph 6. GNU Compiler Efficiency Plot

Analysis and Conclusions:

The performance analysis of the multithreaded Game of Life program reveals key insights into its behavior under different thread counts using both the Intel and GNU compilers. The analysis focuses on execution time, speedup, and efficiency for a 5000x5000 grid over 5000 generations.

For the Intel compiler, the results demonstrate a significant reduction in execution time when increasing the thread count from 1 to 2, showing nearly a perfect speedup. However, beyond 2 threads, the speedup becomes inconsistent, especially for 4 and 8 threads. This fluctuation suggests some overhead in thread management or memory contention. Optimal performance is observed at 10 threads with the least average time. The Intel compiler showcases excellent scaling up to 2 threads, with near-ideal speedup. However, at 4 threads, the speedup declines significantly, hinting at inefficiencies in parallel task allocation or increased synchronization overhead. The program regains efficiency at 10 threads but experiences diminishing returns beyond that point. The efficiency drops as the thread count increases beyond 2. This drop is most pronounced at 4 and 8 threads, indicating that adding more threads does not proportionally improve performance. Nevertheless, 10 threads exhibit a good balance between speedup and efficiency, nearing the ideal parallel performance.

For the GNU compiler, the GNU compiler exhibits a different performance profile. While the time reduction is substantial from 1 to 2 threads, the overall execution time is higher than that of the Intel compiler. Interestingly, the program maintains more consistent performance across all thread counts, with less variation compared to the Intel results. The GNU compiler's speedup is strong at 2 threads but slightly less effective at 4 and 8 threads compared to Intel. However, the GNU compiler consistently performs better at 16 and 20 threads, maintaining speedup close to 2x, showing better scalability in this range. The efficiency results follow a similar trend, with the GNU compiler showing more stable efficiency as thread count increases. While the efficiency is lower than Intel for small thread counts, it becomes more stable at higher thread counts (e.g., 16 and 20), which may suggest better thread management or reduced synchronization overhead for larger grids.

The Intel compiler performs better in terms of raw execution time for small to medium thread counts (2-10), achieving better speedup and efficiency at 10 threads. However, the GNU compiler proves to be more consistent and scalable for larger thread counts, performing more efficiently beyond 10 threads. The optimal thread count for the Intel compiler is 10, whereas for the GNU compiler, 16 and 20 threads show good scalability. In summary, both compilers have their strengths depending on the grid size and thread count. The Intel compiler offers better performance for smaller thread counts, while the GNU compiler is better suited for larger thread configurations. This demonstrates the importance of choosing the appropriate compiler and thread count based on the specific requirements of the workload.

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