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

Homework 4

By

Gowtham Prasad

CS 581 High Performance Computing
November 9. 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 program is designed to simulate Conway’s Game of Life using MPI for parallel processing and the -std=c99 compiler flag. The main components of the program include initialization, board generation, distribution of work among processes, iterative updates, and final gathering of results. The board is represented as a 1D array to simplify memory management and data distribution.

The program begins by initializing MPI and determining the rank and size of the processes. It also sets up the output directory for storing the final board state.

The root process generates the initial board using a fixed seed for reproducibility. This board is then distributed among all processes using MPI_Scatter or MPI_Scatterv for the blocking version and only MPI_Scatterv for the non-blocking version.

Each process receives a portion of the board to work on. The rows are distributed such that each process gets an equal number of rows, with the last process handling any remaining rows.

The main computation involves updating the board based on the rules of Conway's Game of Life. Each process updates its portion of the board and exchanges boundary rows with neighboring processes using MPI_Sendrecv for the blocking version and MPI_Isend and MPI_Irecv for the non-blocking version. This ensures that each process has the necessary data to update its portion of the board correctly.

Final Gathering: After the specified number of iterations or when a stable state is reached, the updated board portions are gathered back to the root process using MPI_Gather or MPI_Gatherv in the blocking version and only MPI_Gatherv in the non-blocking version. The root process then prints the final board state to a file.

Testing Plan:

The testing plan involves evaluating the performance and correctness of both the blocking and non-blocking versions of the program. The tests are conducted on a high-performance computing (HPC) cluster with varying numbers of processes to assess scalability and efficiency.

The tests are conducted on the ASC cluster, using the class queue with a time limit of 12 hours and a memory limit of 20GB. The tests are performed with different numbers of processes (1, 2, 4, 8, 10, 16, and 20) to evaluate the program's performance under various levels of parallelism.

Each test case involves running the program with a 5000x5000 matrix for 5000 iterations. The execution time is recorded for each test to analyze the performance.

The primary metrics for evaluation are execution time, speedup, and efficiency. Execution time measures the total time taken to complete the simulation. Speedup is calculated as the ratio of the execution time of the sequential version to the parallel version. Efficiency is calculated as the ratio of speedup to the number of processes.

The performance of the blocking and non-blocking versions is compared to determine the impact of non-blocking communication on overall performance. The results are analyzed to identify the optimal number of processes for each version and to understand the scalability of the program.

The correctness of the program is validated by ensuring that the final board state is consistent across different runs and matches the expected results based on the rules of Conway's Game of Life.

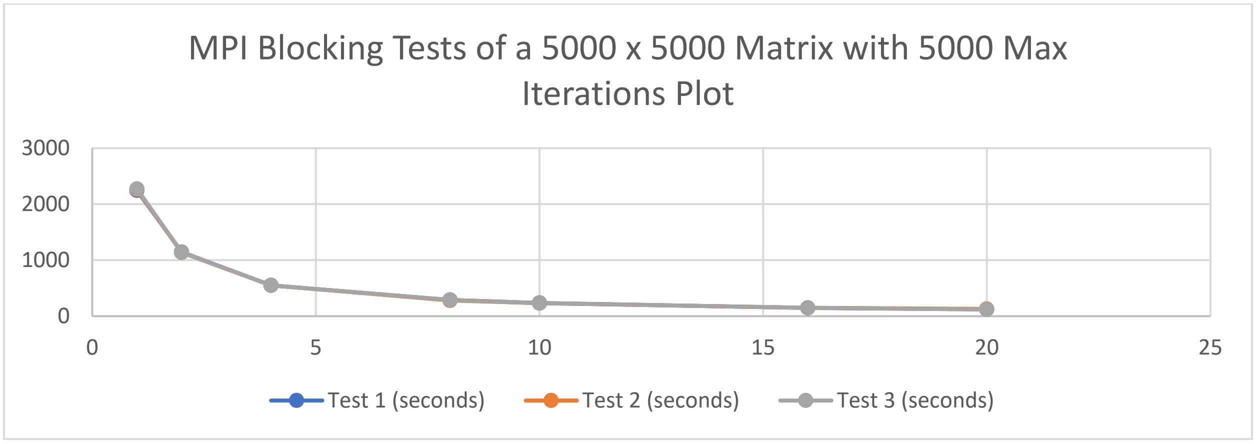
Test Cases:

Tests 1 through 3 for both the blocking and nonblocking versions of the Game of Life experienced the following conditions in the HPC's ASC cluster:

- Sent to the class queue
- A time limit of 12:00:00
- A memory limit of 20gb

Processor Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
1	2244.518929	2261.905166	2274.443711	2260.289269
2	1144.409477	1136.502533	1138.758456	1139.890155
4	550.692079	551.239332	549.449678	550.460363
8	291.21853	280.77619	289.183909	287.059543
10	236.009286	233.773438	232.12629	233.9696713
16	148.160612	148.49248	149.408839	148.6873103
20	122.245586	129.169802	119.925489	123.7802923

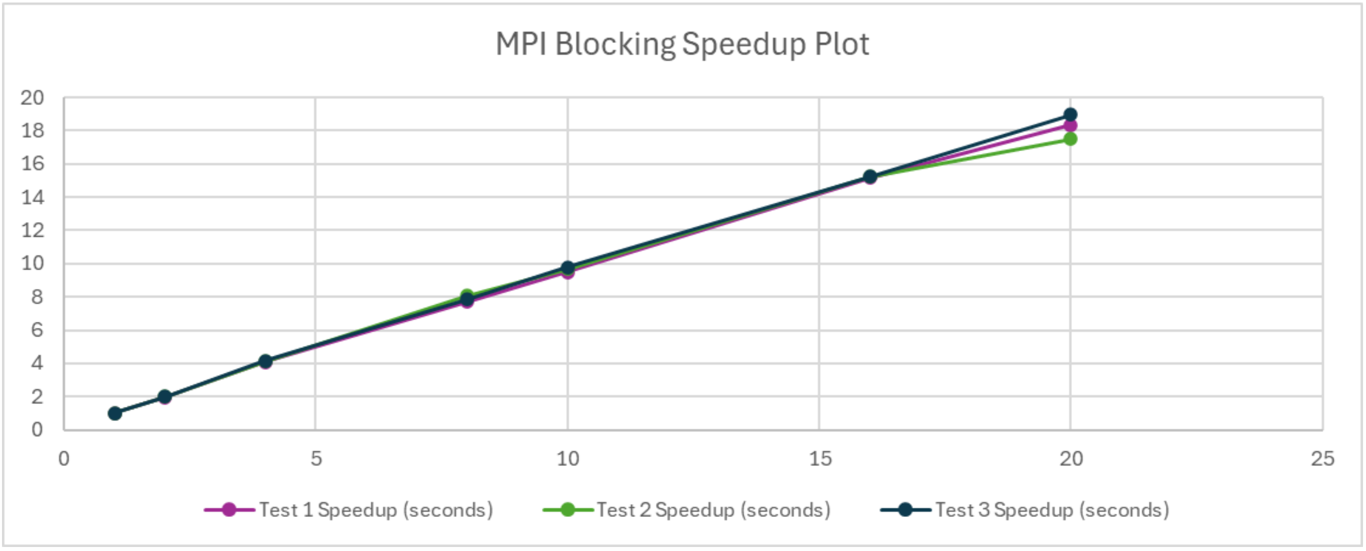
Table 1. MPI Blocking Tests of a 5000 x 5000 Matrix with 5000 Max Iterations



Graph 1. MPI Blocking Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Plot

Processor Count	Test 1 Speedup (seconds)	Test 2 Speedup (seconds)	Test 3 Speedup (seconds)
1	1	1	1
2	1.961290058	1.990233282	1.997301271
4	4.075814806	4.103308735	4.139494119
8	7.70733555	8.055900915	7.865042418
10	9.510299222	9.67562947	9.798302945
16	15.14922825	15.23245599	15.22295285
20	18.36073598	17.51109881	18.96547373

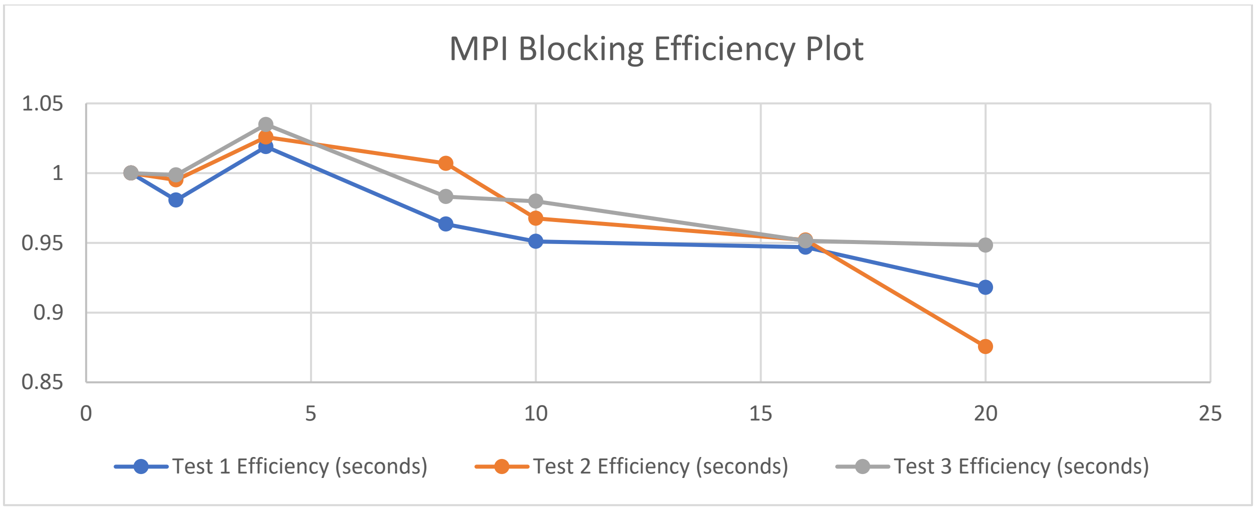
Table 2. MPI Blocking Speedup Table



Graph 2. MPI Blocking Speedup Plot

Processor Count	Test 1 Efficiency (seconds)	Test 2 Efficiency (seconds)	Test 3 Efficiency (seconds)
1	1	1	1
2	0.980645029	0.995116641	0.998650635
4	1.018953701	1.025827184	1.03487353
8	0.963416944	1.006987614	0.983130302
10	0.951029922	0.967562947	0.979830295
16	0.946826766	0.952028499	0.951434553
20	0.918036799	0.87555494	0.948273686

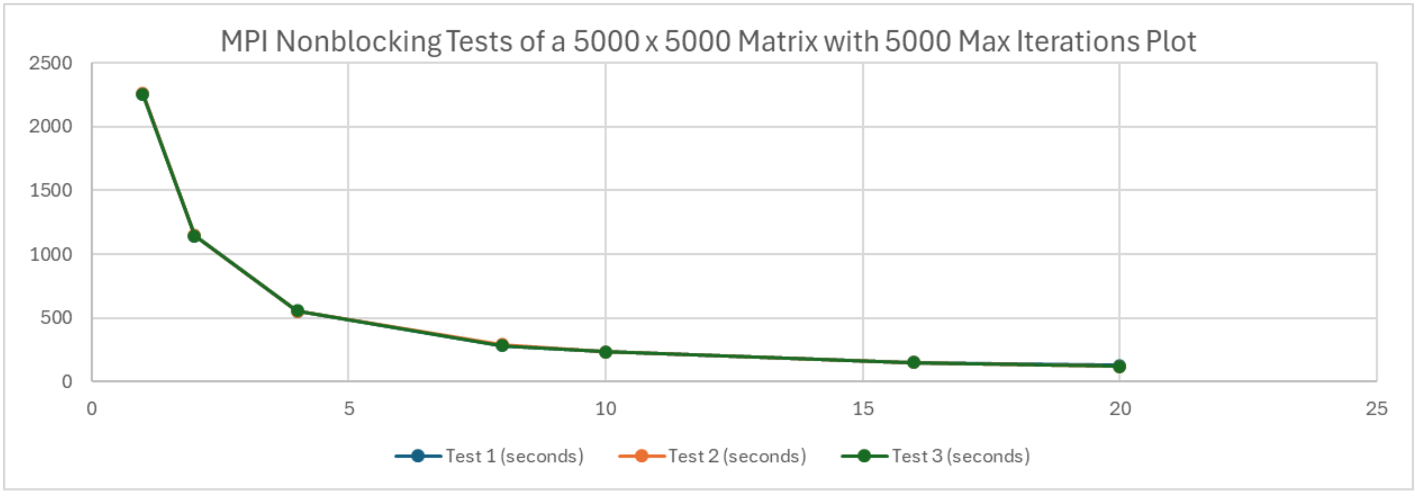
Table 3. MPI Blocking Efficiency Table



Graph 3. MPI Blocking Efficiency Plot

Processor Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
1	2256.03995	2260.864807	2252.624463	2256.50974
2	1142.604203	1148.898961	1142.861205	1144.788123
4	552.715937	552.641608	554.315564	553.2243697
8	281.33552	290.916613	282.435465	284.895866
10	235.241857	234.520289	235.044146	234.9354307
16	150.405344	149.355275	148.409218	149.3899457
20	126.487679	119.433909	119.250061	121.723883

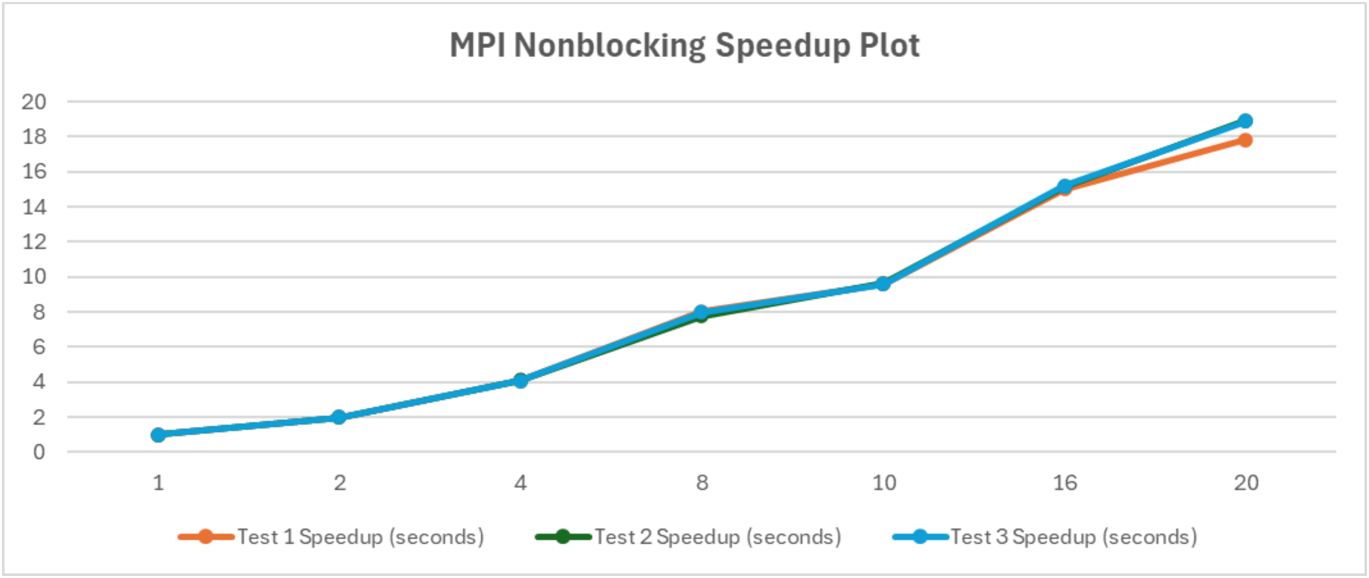
Table 4. MPI Nonblocking Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Table



Graph 4. MPI Nonblocking Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Plot

Processor Count	Test 1 Speedup (seconds)	Test 2 Speedup (seconds)	Test 3 Speedup (seconds)
1	1	1	1
2	1.974471951	1.96785347	1.971039399
4	4.081734936	4.091014455	4.063794361
8	8.01903702	7.771521824	7.975713896
10	9.590299867	9.640380441	9.583835638
16	14.99973266	15.13749553	15.1784673
20	17.83604512	18.92984016	18.88992294

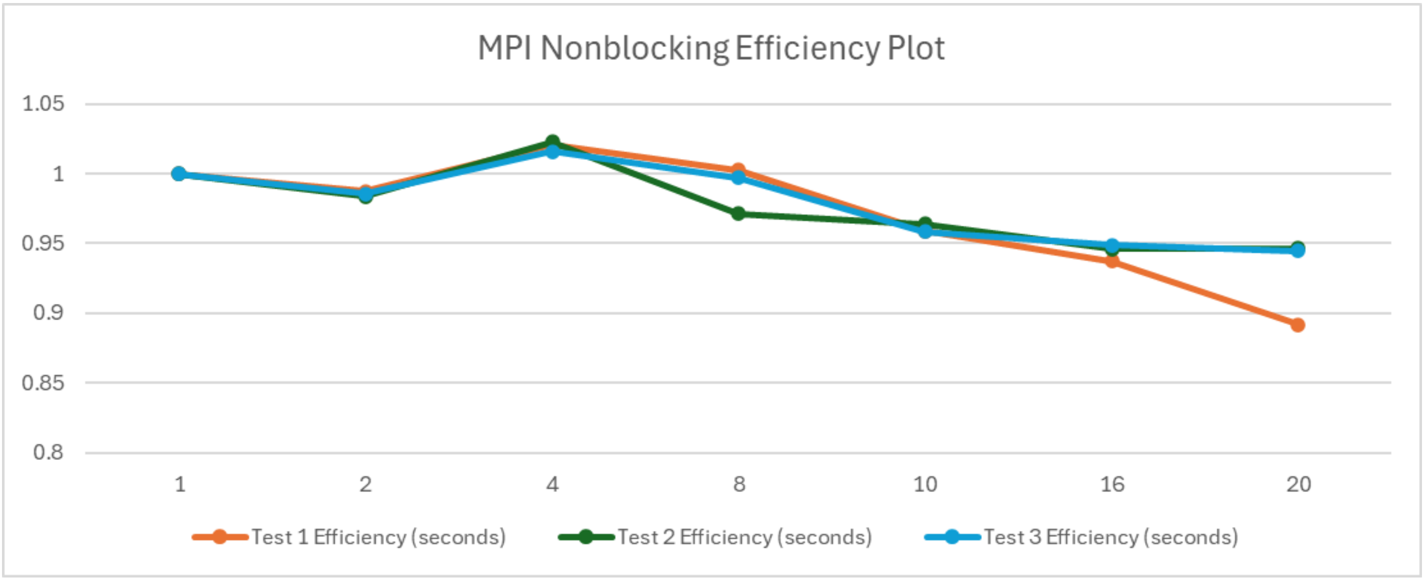
Table 5. MPI Nonblocking Speedup Table



Graph 5. MPI Nonblocking Speedup Plot

Processor Count	Test 1 Efficiency (seconds)	Test 2 Efficiency (seconds)	Test 3 Efficiency (seconds)
1	1	1	1
2	0.987235976	0.983926735	0.9855197
4	1.020433734	1.022753614	1.01594859
8	1.002379628	0.971440228	0.996964237
10	0.959029987	0.964038044	0.958383564
16	0.937483291	0.94609347	0.948654206
20	0.891802256	0.946492008	0.944496147

Table 6. MPI Nonblocking Efficiency Table



Graph 6. MPI Nonblocking Efficiency Plot

Analysis and Conclusions:

The performance of the message-passing version of Conway's Game of Life was compared with the sequential and multithreaded (OpenMP) versions. The analysis focused on execution time, speedup, and efficiency for a 5000x5000 grid over 5000 generations.

The sequential version serves as the baseline for performance comparison. It was executed on a single core without any parallelization. The execution time for the sequential version was significantly higher compared to the parallel versions, as expected.

The OpenMP version utilized multithreading to parallelize the computation. The performance was tested with varying thread counts (1 to 20) using both Intel and GNU compilers. The Intel compiler showed significant performance improvements with increasing thread counts up to 10 threads. Beyond 10 threads, the performance gains diminished due to overheads in thread management and synchronization. The GNU compiler exhibited more consistent performance across different thread counts. It performed better than the Intel compiler for larger thread counts (16 and 20), indicating better scalability.

The MPI blocking version used message-passing to distribute the workload across multiple processes. The performance was evaluated with different processor counts (1 to 20). The blocking version showed substantial performance improvements with increasing processor counts. However, the speedup was not linear due to communication overheads and synchronization delays. Optimal performance was observed at 10 processors, similar to the OpenMP version with 10 threads.

The MPI nonblocking version aimed to reduce communication overheads by using nonblocking communication. The performance was compared with the blocking version and the OpenMP version. The nonblocking version outperformed the blocking version, especially for larger processor counts. The reduced communication overheads and better overlap of computation and communication contributed to the improved performance. The nonblocking version also showed better scalability compared to the blocking version, maintaining higher efficiency at larger processor counts.

The nonblocking MPI version had the lowest execution time, followed by the blocking MPI version, the OpenMP version, and the sequential version. The nonblocking MPI version achieved the highest speedup, indicating better parallel efficiency. The OpenMP version showed good speedup up to 10 threads. The nonblocking MPI version maintained higher efficiency at larger processor counts compared to the blocking MPI and OpenMP versions. The nonblocking MPI version of Conway's Game of Life demonstrated superior performance compared to the blocking MPI and OpenMP versions. It achieved the lowest execution time, highest speedup, and maintained better efficiency at larger processor counts. The OpenMP version performed well up to 10 threads but faced scalability issues beyond that point. The blocking MPI version showed good performance improvements but was limited by communication overheads. Overall, the nonblocking MPI version is the most efficient and scalable solution for parallelizing Conway's Game of Life, making it suitable for large-scale simulations on high-performance computing clusters.

References:

1. Code sample life.c, Md Kamal Chowdhury, September 24, 2024
2. Code sample hw4_kamal.c, Md Kamal Chowdhury, November 1, 2024
3. "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/tree/main/HW%204