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 uses 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.

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.

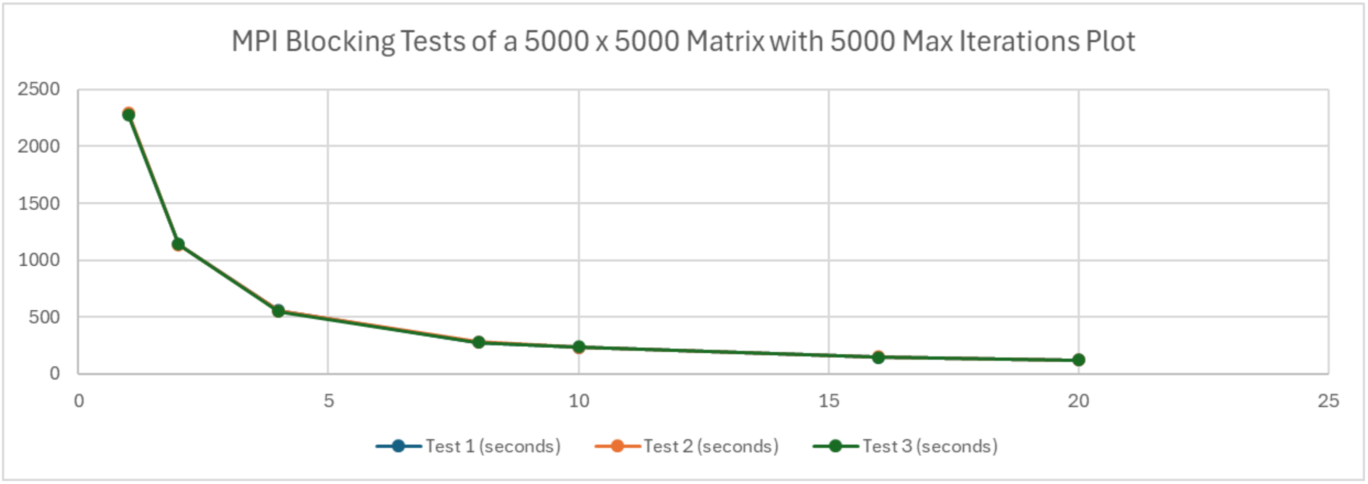
Test Cases:

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	2284.79683	2294.894456	2275.254793	2284.982026
2	1137.775183	1135.961816	1139.198702	1137.645234
4	560.125051	551.893885	550.388498	554.1358113
8	276.821429	280.987266	278.128223	278.6456393
10	233.298551	234.92261	237.612626	235.277929
16	149.00259	150.632845	148.448597	149.361344
20	121.318126	121.623063	121.533316	121.4915017

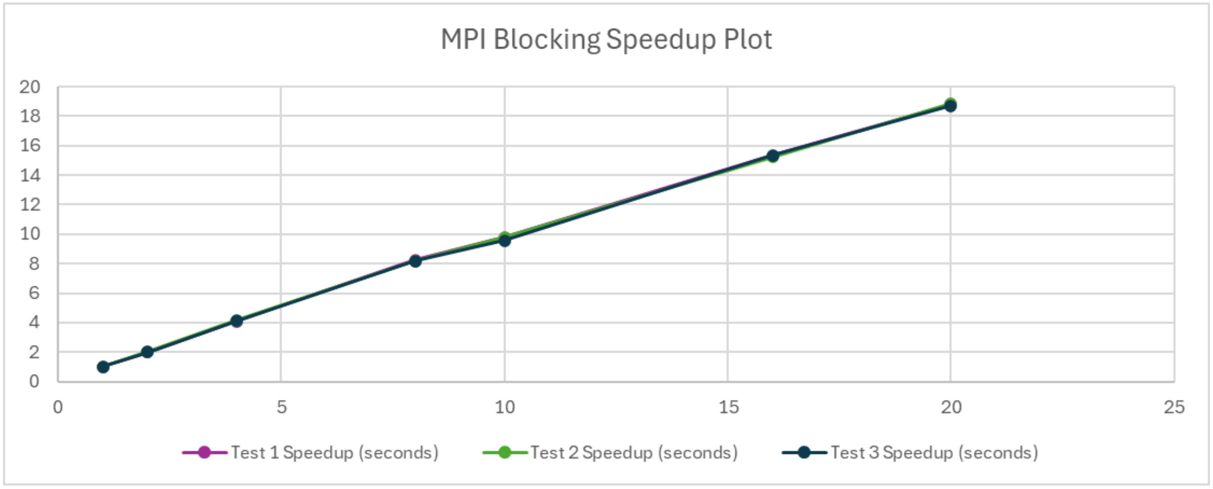
Table 1. MPI Blocking Tests of a 5000 x 5000 Matrix with 5000 Max Iterations



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	2.008126794	2.020221475	1.997241385
4	4.079083458	4.158216857	4.13390687
8	8.253684833	8.167254298	8.180596591
10	9.793446295	9.768725352	9.575479348
16	15.33394037	15.23502033	15.32688647
20	18.83310355	18.86890857	18.72124342

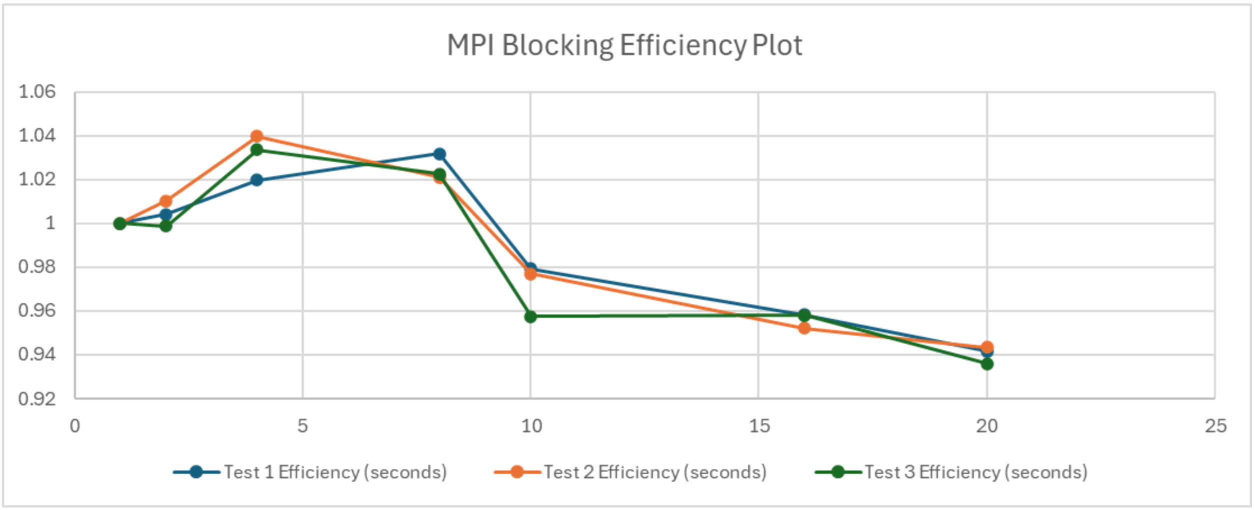
Table 2. MPI Blocking Speedup Table



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	1.004063397	1.010110738	0.998620692
4	1.019770865	1.039554214	1.033476717
8	1.031710604	1.020906787	1.022574574
10	0.97934463	0.976872535	0.957547935
16	0.958371273	0.952188771	0.957930404
20	0.941655178	0.943445429	0.936062171

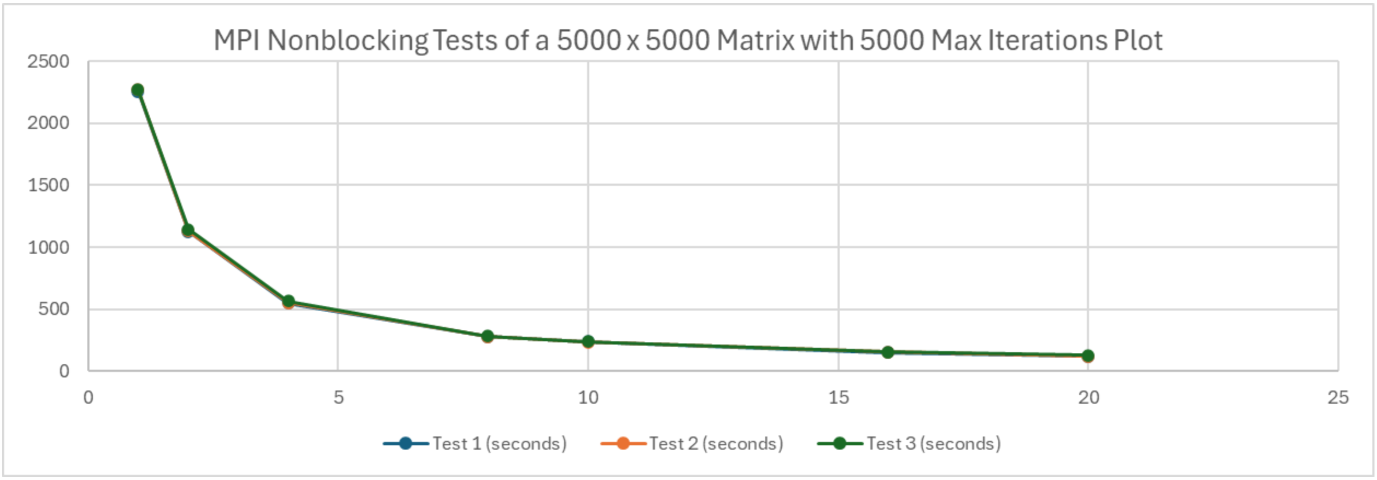
Table 3. MPI Blocking Efficiency Table



Graph 3. MPI Blocking Efficiency Plot

Processor Count	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
1	2255.976154	2269.124498	2267.936777	2264.34581
2	1127.037527	1131.636482	1143.80556	1134.159856
4	548.102434	552.792732	564.631244	555.17547
8	279.109187	278.211891	282.032497	279.784525
10	237.67192	233.998009	236.793498	236.1544757
16	148.765085	153.167848	154.516396	152.1497763
20	120.707381	120.783352	129.688689	123.726474

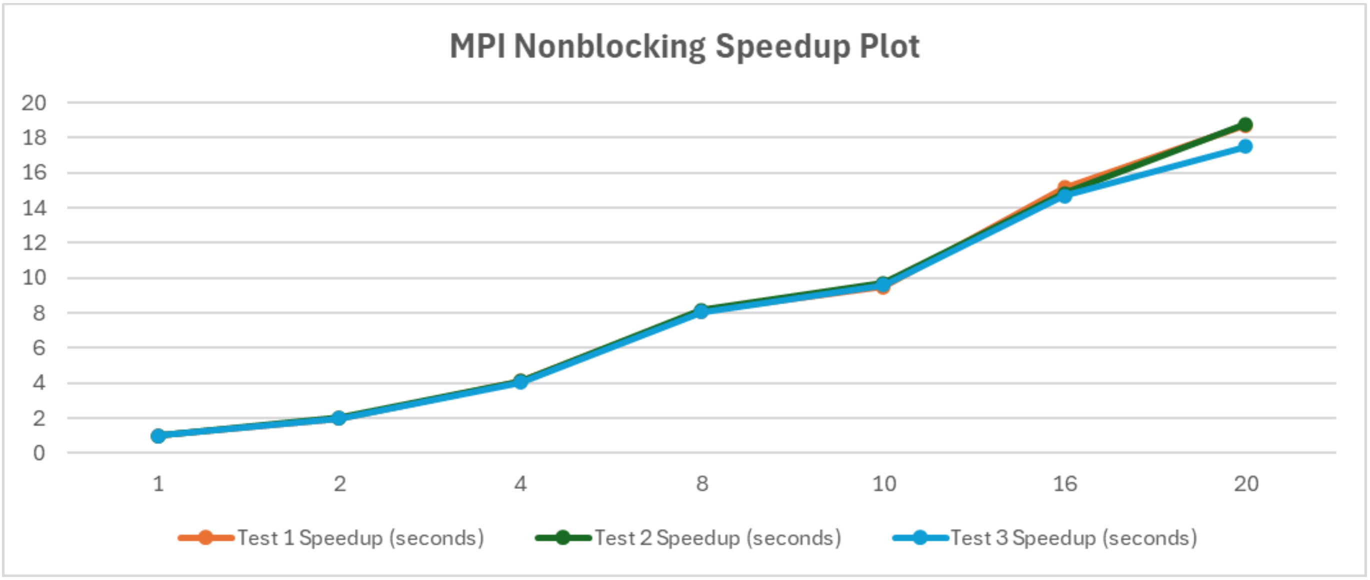
Table 4. MPI Nonblocking Tests of a 5000 x 5000 Matrix with 5000 Max Iterations Table



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	2.001686812	2.00517086	1.982799224
4	4.115975435	4.104837793	4.016668934
8	8.082772833	8.156101775	8.041402325
10	9.491975973	9.697195748	9.577698696
16	15.16468837	14.81462675	14.6776448
20	18.68962888	18.78673228	17.48754494

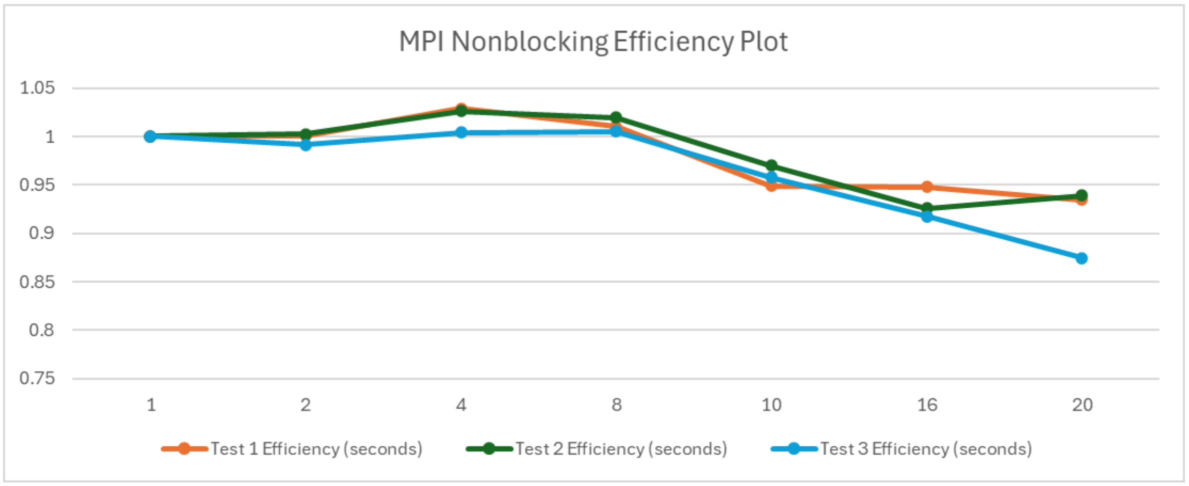
Table 5. MPI Nonblocking Speedup Table



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	1.000843406	1.00258543	0.991399612
4	1.028993859	1.026209448	1.004167234
8	1.010346604	1.019512722	1.005175291
10	0.949197597	0.969719575	0.95776987
16	0.947793023	0.925914172	0.9173528
20	0.934481444	0.939336614	0.874377247

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