PARALLEL IMPLEMENTATION OF CONWAY’S GAME OF LIFE

Homework # 3

By

Nolan McKivergan

CS 581 High Performance Computing
November 7, 2024

GitHub Link: https://github.com/npmckivergan/Fall2024_CS581_HW4

Problem Specification:

Conway's Game of Life is a cellular automaton invented by mathematician John Conway in 1970. It is a zero-player game, meaning the progression is determined purely by the initial input. The game is inspired by the motion of biological colonies over many generations and is used in a wide variety of fields ranging from biology, to computer science, to cryptography.

The game is played on a square two-dimensional grid, where each cell can be in one of two states: dead or alive. For each generation, every cell in the grid is updated simultaneously according to a set of rules regarding the status of the neighbors. Each cell has eight neighbors since diagonals are included.

The rules determining whether a cell becomes alive or dead are as follows:

1. If the cell is alive:
    a. If the cell has less than two living neighbors, the cell dies from loneliness.
    b. If the cell has more than three living neighbors, the cell dies from overpopulation.
    c. If the cell has exactly two or three living neighbors, it remains alive.
2. If the cell is dead:
    a. If the cell has exactly three living neighbors, it becomes alive from reproduction.
    b. Otherwise, the cell remains dead.

This game was to be implemented using C/C++, with the grid size, maximum number of generations, number of processes, and output file location passed in as command-line arguments. The cells were to be initialized to alive or dead states randomly at the start of the program. The program runs without printing anything and terminates when there are no changes between two subsequent boards or the maximum number of generations has been reached. This particular implementation must utilize parallel computing to improve performance. Regions of the board are evaluated in parallel using MPI for dividing the workload among multiple processes. Upon termination, the program prints out the total time taken to run the game, the final generation number, and writes the final board to an output file.

Two version of this program were to be implemented: one using the standard MPI_Sendrecv function, and the other using non-blocking, point-to-point MPI calls to exchange data between the processes with functions such as MPI_Isend, MPI_Irecv, MPI_Wait, and MPI_Test.

Program Design:

Both programs were implemented using standard C to maximize performance and memory efficiency. MPI was used to facilitate parallel computation with multiple processes.

Both programs had the same core functionality which is laid out below.

Based on the provided size of the grid, the workload was partitioned by rows, and each process dynamically allocates two two-dimensional arrays to hold the local matrix, with one extra row and column to allow for ghost cell boundaries and overlap with processes holding the neighboring rows. Two matrices are used so one can statically represent the previous generation, while another can receive the changes made by the current generation.

Following creation, every element among the local matrices is initialized using values generated by the rand function, which produces a randomized but repeatable sequence of 1's and 0's to represent live and dead cells.

MPI_Scatterv is used to scatter the partitioned workload to each individual process. Following this, each process enters the main generation loop.

Within the main loop, the two programs differed slightly. In the first version, MPI_Sendrecv is directly used to exchange both the first and last rows of each processes local matrix with the process holding the neighboring matrix in one call. However, in the second implementation, two calls are made to both MPI_Isend and MPI_Irecv to manually send and receive the first and last rows. Then, a call to MPI_Waitall is used to block execution until all processes have successfully exchanged information. Execution in both programs is the same after this point, with each process applying the Game of Life rules to its local matrix and pushing the changes to the future matrix. Upon completion of the generation, the current and future matrix poin ters are swapped. Provided some changes were detected during the computation, the loop repeats. Otherwise, the loop breaks since there were no changes between two subsequent generations.

After exiting the computation, the final state of the board is gathered from all processes using MPI_Gatherv. This final board is printed to the output file, and the total time taken to complete the generation loop and total number of generations is printed to the console.

Testing plan:

This program had to be tested for both correctness and performance. These test cases were performed separately and are detailed below.

To test for correctness, I ran several trials of both programs and printed the final boards to files. Both programs computed a 1000x1000 board with 5000 maximum generations for 1, 2, 4, 8, 16, and 20 processes. To obtain a control, I ran my serial implementation of the game of life with the same board size and printed the output to a file. Comparing all resulting output files with the Linux command “diff” will determine if these outputs are identical. If they are, it will serve as proof that the program functions the same for different numbers of processes.

To test for performance, test cases of a grid of 5000x5000 was iterated 5000 times with 1, 2, 4, 8, 16, and 20 processes on the ASC Cluster with both programs. Each test case was run three times so an average can be used. To allow these processes to effectively parallelize the workload, 20 cores and 64 GB of RAM were allocated to the job. The time taken to compute the board to completion was recorded, and the results are shown in the analysis section.

For the performance test, it is important to request twenty CPU cores in the job submission to allow for multiprocessing to actually speed up computation. If there are fewer cores than processes, the operating system will schedule the processes to virtualize having additional CPUs and allow the program to run, but the total execution time will not improve without separate computing units running in parallel.

Test Cases:

To test for correctness, I used the “correctness.sh” script to run the test cases, produce the output files, and check for differences between all of them. In the “correctness” job output file, nothing was printed out by the “diff” command, indicating no difference was found between any of the boards. This shows that the final boards are identical for 1, 2, 4, 8, 16, and 20 processes on both MPI programs and the serial program, which serves as evidence of the correctness of the program.

To test performance, I ran both programs with a 5000x5000 board and trials of 1, 2, 4, 8, 16, and 20 processes for 5000 iterations. Each test case was run three times and the average time was computed. Twenty CPU cores are requested in the job submission for the reasons laid out in the testing plan section. The performance.sh script was used to run the programs, and the results were captured in the job output file. The recorded times are shown in the table below for both programs, along with the speedup factor compared to the single-process run.

In addition, the OpenMP time trials from project 3 are also shown to illustrate the performance differences between the two parallelization tools.

MPI Programs

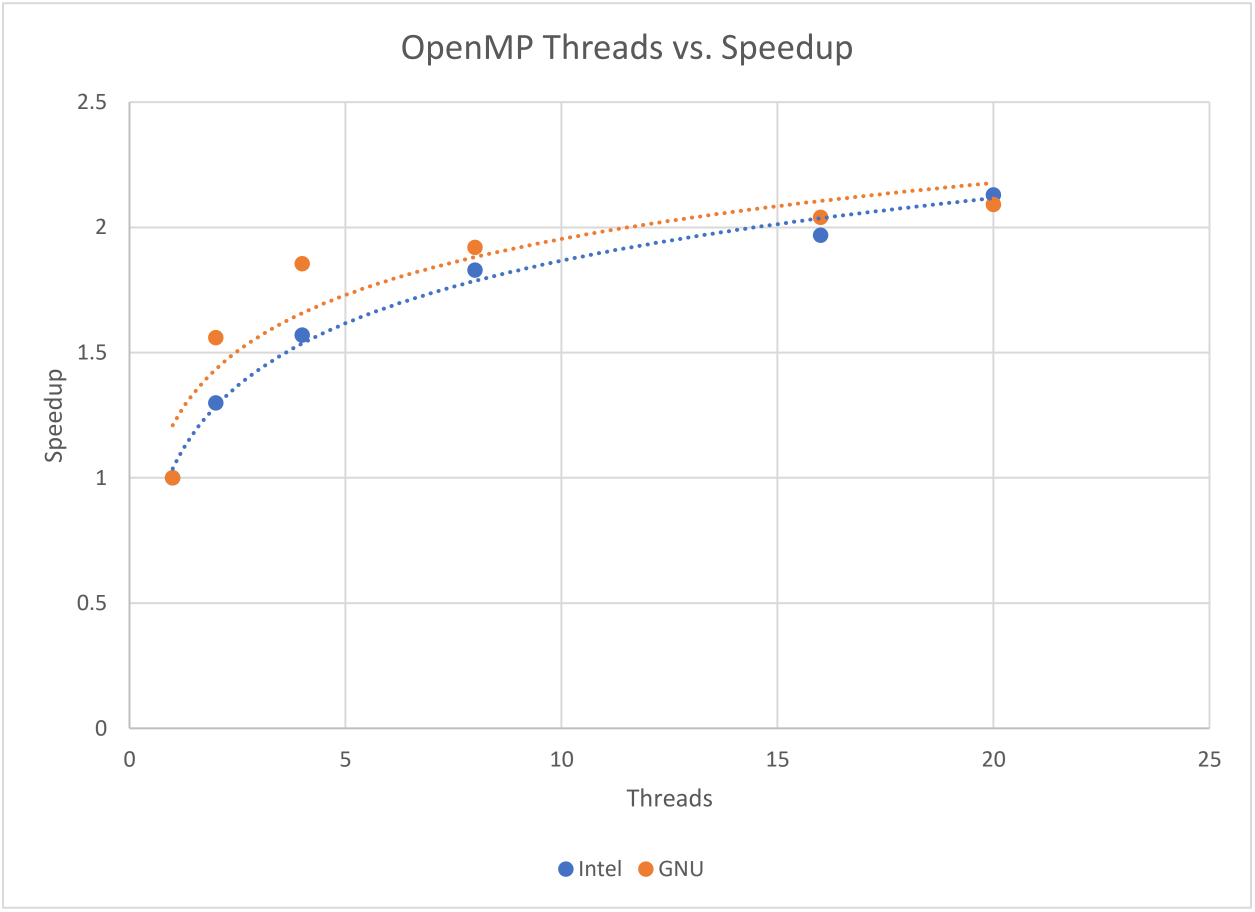
Processes	Program 1 Time (s)	Program 2 Time (s)	Program 1 Speedup	Program 1 Efficiency	Program 2 Speedup	Program 2 Efficiency
1	236.855	245.419	1	1	1	1
2	118.999	122.896	1.99	1	2	1
4	59.471	61.468	3.98	1	3.99	1
8	30	30.96	7.9	0.99	7.93	0.99
16	15.97	16.32	14.83	0.93	15.04	0.94
20	14.099	13.3481	16.8	0.84	18.39	0.92

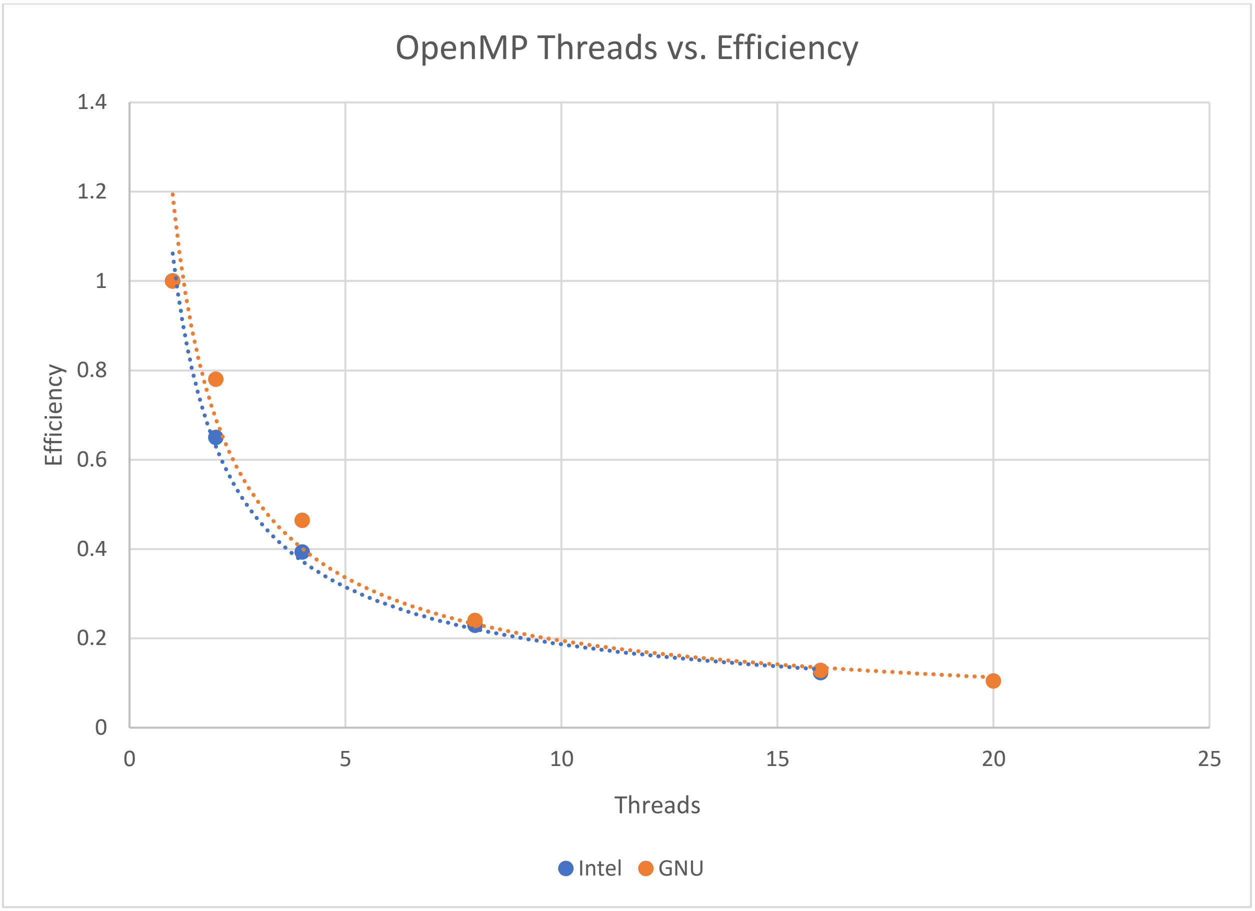
OpenMP Program

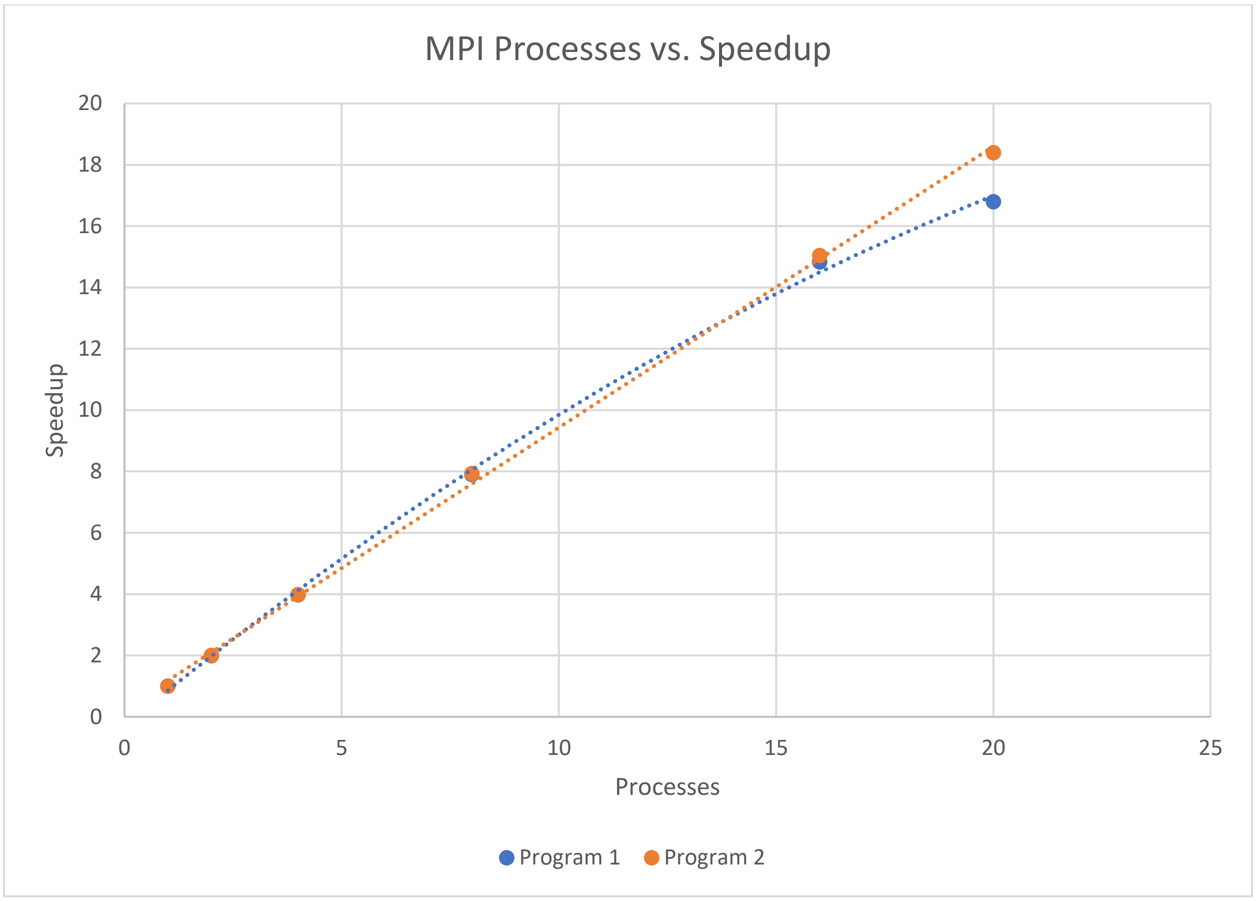
Threads	Intel Time	GNU Time	Intel Speedup	GNU Speedup	Intel Efficiency	GNU Efficiency
1	431.712	430.431	1	1	1	1
2	332.142	276.104	1.3	1.56	0.65	0.78
4	274.951	232.036	1.57	1.855	0.393	0.464
8	235.886	224.214	1.83	1.92	0.229	0.24
16	219.237	210.956	1.969	2.04	0.123	0.128
20	203.097	205.819	2.13	2.091	0.101	0.105

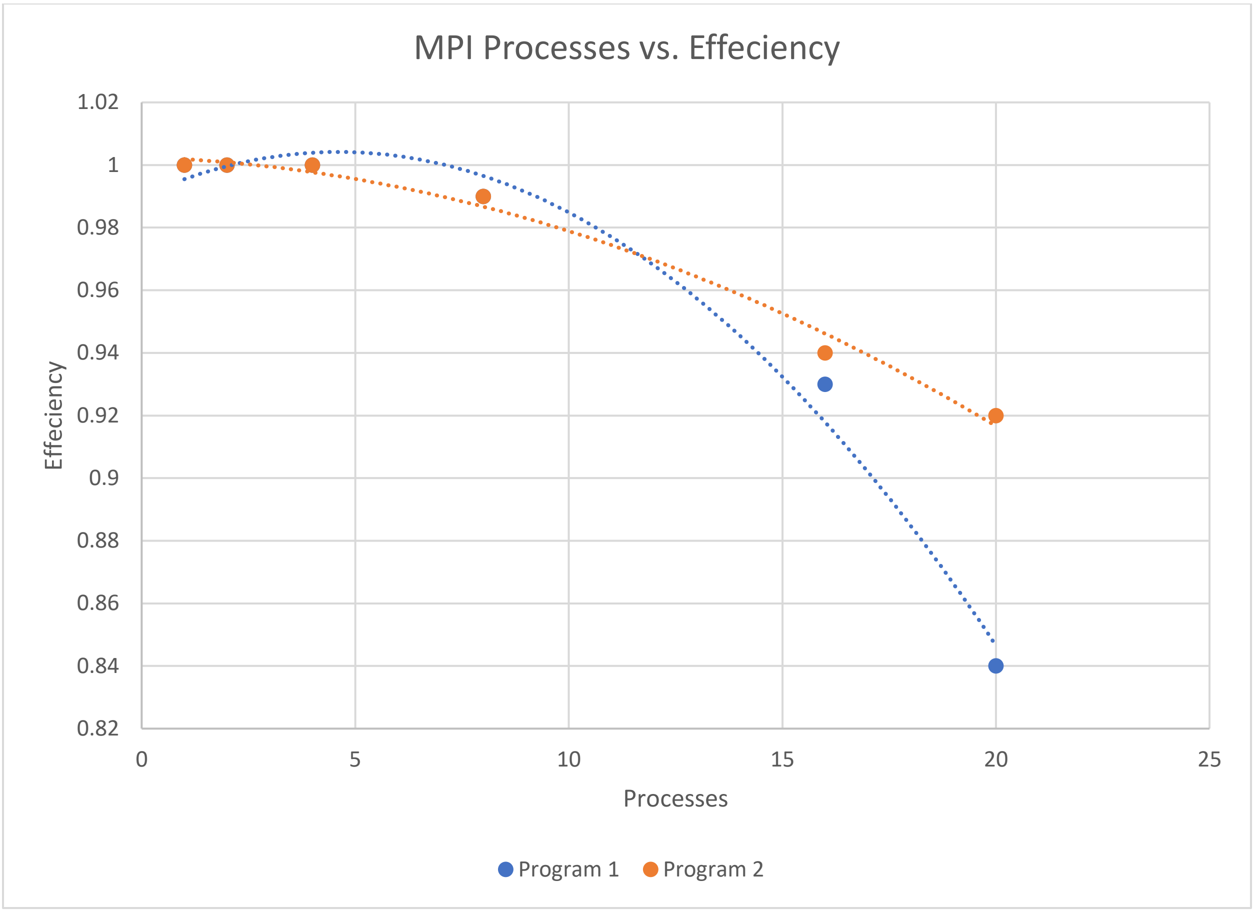
Analysis and Conclusion:

The following plots shows the relationship between the number of processes and the calculated speedup and efficiency factors of the execution of the program for the same grid size and total number of allocated cores. Results are shown for both the MPI and OpenMP implementations of the Game of Life.




OpenMP Threads vs. Efficiency
Efficiency
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
Threads
● Intel ● GNU


MPI Processes vs. Speedup
Speedup
20
18
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
Processes
Program 1
Program 2



These graphs show that both MPI programs experiences significantly greater gains in performance when run with multiple processes. In the MPI Processes vs Speedup graph, it is apparent that the speedup factor increases along with the number of processes in a near-linear fashion across both MPI programs. This suggests that additional processes allocated to the program are reducing the total runtime close to as much as physically possible.

However, the MPI Processes vs Efficiency graph shows a slight but nontrivial dip in efficiency on a per-process basis as the number of processes approaches the total number of available cores allocated to the job. This could be due to internal workings of the operating system, as it may take longer to schedule the processes among available cores as the resources become more strained.

In both MPI graphs, while the performance metrics were similar between the two programs, program 2 (the non-blocking point-to-point information passing implementation) exhibited slightly stronger performance over program 1, which used the built-in MPI_Sendrecv function to pass data to neighboring processes. This is likely because MPI_Sendrecv is an expensive overhead operation with many working parts behind the scenes. By directly making the non-blocking point-to-point calls directly where they are needed, we can avoid unnecessary blocking of processes and achieve slightly higher performance.

However, both MPI programs convincingly outperfromed the OpenMP program, achieving much better speedup factors and process efficiency. The most likely reason for this is because the ASC Cluster is a distributed-memory setup, where each node has its own memory. MPI is optomized for distriuted-memory machines. On the other hand, OpenMP is engineered to perform best on shared-memory devices, such as a PC with multiple CPU cores but only one bank of RAM. As such, given the distributed-memory environment on which these programs were tested, it follows that the programs using MPI had much better performance.

Questions:

1. Did you achieve the goals or objectives of this assignment?

   Yes. I created an efficient and accurate implementation of Conway's Game of Life utilizing MPI for parallelization, which was the specific objective of this assignment.

2. How did you achieve these goals (be specific)?

   I studied online resources to develop a solid understanding of the game. I read the assignment document to understand how I was to handle all the small nuances of the game for this purpose. I became familiar with the MPI API and how to properly synchronize processes and exchange information between them. I carefully wrote a C program to implement the game with multiprocessing, and during that process, I thoroughly tested my code to ensure it was doing exactly what I wanted and properly parallelizing the computations. Afterward, I rigorously tested my final program for correctness and performance with my own test cases and those provided in the assignment document.

3. What can you infer from the performance results you have obtained? Does the performance results match your expectation? Explain.

   As I said in the analysis section, based the results we can infer that MPI is much more effective at parallelizing tasks on a distributed-memory system than OpenMP. Furthermore, by using non-blocking end-to-end calls to exchange information among the processes rather than the built-in MPI_Sendrecv functions, even better performance is possible. Using MPI, tasks on a distributed system such as the ASC Supercomputer Cluster can be parallelized with an almost perfect degree of efficiency provided there are as many or more physical cores allocated to the job as processes in the program.

4. Looking back on the project what would you do differently?

   If I were to do this project again, I would start sooner. Implementing multiprocessing was difficult, and it took a lot of time and effort to get it working properly and test it for correctness.

5. What changes would you make in the design phase of your project?

   During the design phase, I would take greater care to properly get the processes to communicate and synchronize before trying to implement in the code. It took me several tries to implement it in code, and I spent a lot of time just trying to change random factors to get it to work. A more design-oriented approach would have been easier in the long run.

6. What additions would you suggest to improve the working of your program?

   Honestly, I don’t know of any way to improve my program from a design perspective. I properly implemented multithreading in my program and verified functionality and correctness. It works and it works well. I don’t see any way it can be drastically improved while remaining a C program that runs on standard computer cores. Using specialized hardware such as GPUs for parallel processing would likely improve performance, but that would be a major overhaul.

References:

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

https://playgameoflife.com/

https://hpc.nmsu.edu/discovery/mpi/mpi-openmp/#:~:text=With%20MPI%2C%20each%20process%20has,no%20notion%20of%20message%2Dpassing.

https://www.asc.edu/service-area/high-performance-computing

Homework 4 Helper PowerPoint from BlackBoard