Analysis of Conway's Game of Life with OpenMPI Implementation.

John Sabo
Parallel and Concurrent Programming
Rowan University

April 26 2024

Abstract

This paper presents an analysis of Conway's Game of Life implemented with OpenMPI parallelization. The program, written in C, utilizes nine functions to simulate the game. OpenMPI implementation divides the main grid in even amounts of rows between each of the processes, significantly enhancing performance. Verification tests show there is a logical error within the OpenMPI implementation as the outputs of the sequential and parallelized versions are not identical. Performance evaluations reveal substantial speedup for larger matrix sizes with increased process count, however, at a certain process count, due to overhead associated with exchanging ghost cells, the performance beings to diminish.

I. Program Design

The Conway's Game of Life script is written in the C Programming Language and contains nine functions. First, the 'allocArray' function is used to allocate memory for the arrays that will be used in the program. There are several arrays used throughout the program, one for the initial grid, two local grid unique to each process, one for the current generation and one for the next, and a final grid where all of the processes will send their data. Second, the 'printArray' is used specifically for debug and testing purposes. It is used in the 'test' script in the two versions of the program to output all boards created during a run that will then be verified with the 'diff' command. Third, the 'initRandomArray' function is used to populate the initial current array with all random values of either 1.0 or 0.0. Fourth, the 'ghostFill' function is used to copy the second and second to last column and row into the last and first column and row respectively. This allows for easier access when checking the neighbors in each cell of the grid through each generation. Fifth, the 'updateCell' function is used to individually update one cell. It is called N by M times for each generation and it calculates the sum of the neighbors and uses Conway's Game of Life original logic to determine the state of the current cell in the next generation. Sixth, the 'step' function is used to loop through every cell and call the 'updateCell' function. It also determines whether or not there was an update anywhere on the board in the current generation. Seventh, the 'sendReceiveGhostRows' will using MPI's send and receive function to communication each processes boarder rows with its neighboring processes. After communicating the ghost rows, the function will then clean up each processes local grid by ensuring the columns then have the appropriate values. This part of the function requires no aid from MPI as all of the information should be available directly in each process. Eighth, the 'free2DArray' will simply free the memory of a 2D array that is not currently being used. Finally, the main function contains the initialization of all of the important variables such as the current and previous arrays, the N and M dimensions of the board, and the max generation. The main function contains the initialization of MPI and its communicator. The program first creates and populates a two dimensional array of desired size and will randomly generate ones or zeros for each index.

This array is then scattered to each of the processes. I believe this is the part of the program where an issue arises as the correct data is not placed in the correct indexes in the receiving array. This leads to differing output between the sequential version and parallelized version. After it scatters the data to all of the processes, it will then convert the one dimension array the data was received in into a two dimensional array. From here we can run the actual game. Each generation we must communicate the ghost rows with the appropriate function and then run one iteration of the step function. Once there are no changes or the max generation has been reached the program will then break out of the while loop and convert the 2D array back into a 1D array in order to gather the data. The data is gathered into the final array which can be printed. The time taken for the execution of this program is started just before the scatter and stopped immediately after the gather.

Instead of manually running the program over and over again for a variety of test cases and calculating averages in a spreadsheet, I opted to create a bash script that would run a series of commands a certain amount of iterations for each command and compute the average before then dumping it into a text file. In the first assignment, I was more naive and chose to calculate the averages myself leading to a lot more headache. Using a script to get my data proved to be the much less redundant method. Getting my data this way enabled me to write a python script to simply read the generated file with the averages for each command and plot the data using the matplotlib library.

II. Test Process and Planning

I was unable to verify that the output of the sequential version and parallelized version of my program were identical. I believe this to be for a number of reasons. I attempted to run a simple four by four grid for both versions and even when I did one generation, they were still different. I am unsure what is different between them in the first few steps of the program. The initialization and population of the grids, I used the same seed for my random number generation and still produced different results. I believe however, that even if they were identical at this step, the distribution of data is flawed in my implementation. The initial grid should be divided evenly and placed in specific indexes of the receiving array in each process, I was unable to figure out how to accomplish this. Once the data was received in each process, I would need to share ghost cell borders with neighboring processes and I am relatively confident in this portion of the program as I read through the docs vigorously. Another possible flaw in the program has to do with the final gather step. It is possible that I may be overlapping and overwriting data producing some strange patterns in the final outputs.

Regardless of my inability to verify the integrity of my MPI implementation, I still chose to test its performance. I reused the bash script I wrote for assignment two and modified it accordingly. Because I would need a number of process that could evenly divide my total number of rows, I used a 1024 by 1024 board for each of my tests and I used 2,4,8,16,32,64,128, and 256 process to ensure the data would always be divided evenly. My performance data showed that whatever my program was doing, was still speeding up with a larger number of processes. However, at a certain threshold, the overhead associated with the ghost row sharing began to effect the performance because there were so many processes sharing data between their neighbors.

Visualizing the data produced would be straight forward with Python and the matplotlib library. One graph was the efficiency graph and displayed the time taken versus the process count. Ideally we hope to see a negative trend in the graph. The second graph was the speed up graph which showed how many times faster the program was for each process count when compared to one process, or the sequential version.

III. Data

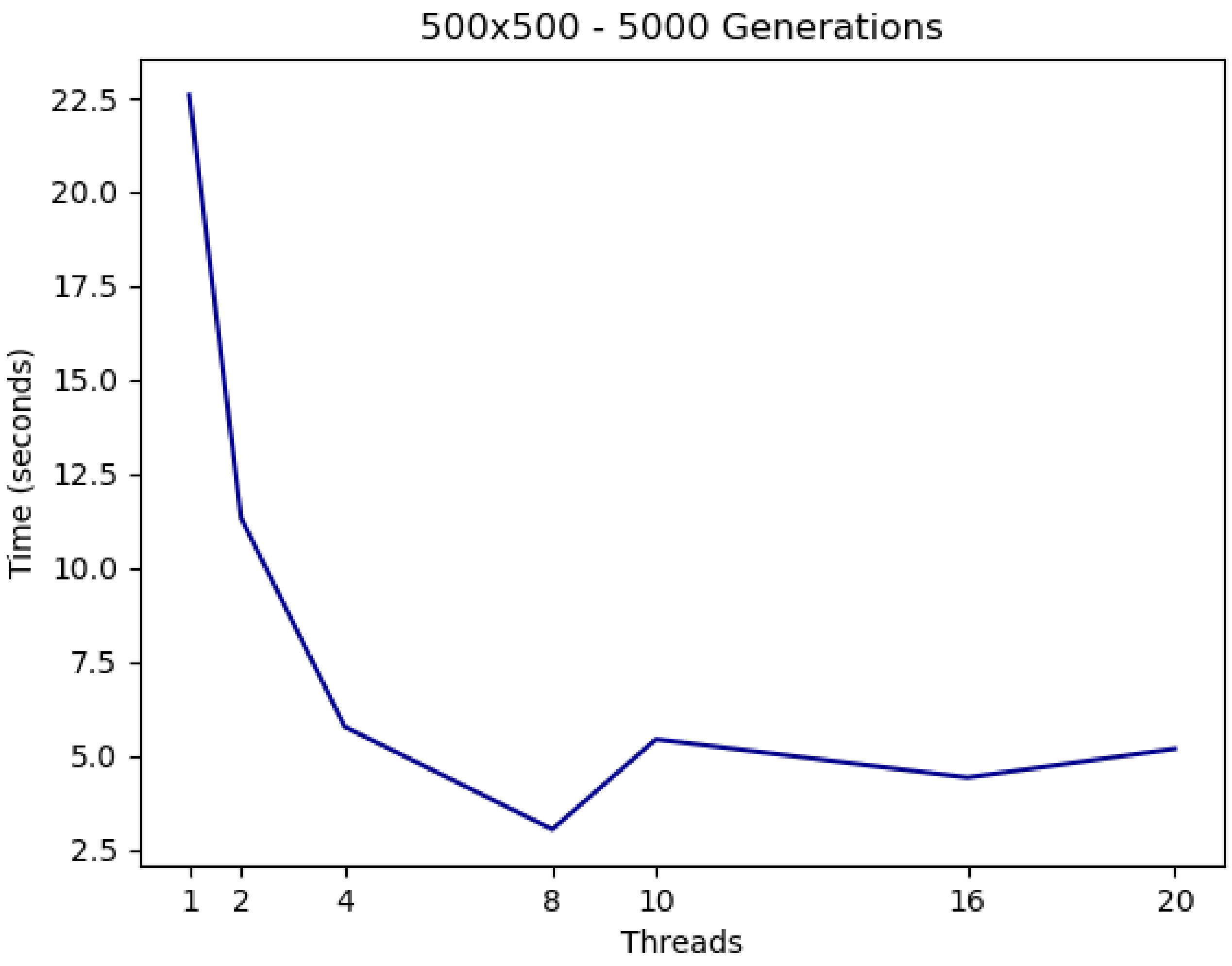

Figure 1: *This is the efficiency graph for the five hundred by five hundred grid ran for five thousand generations with OpenMP. Each thread count was ran for five iterations to determine its average value.*

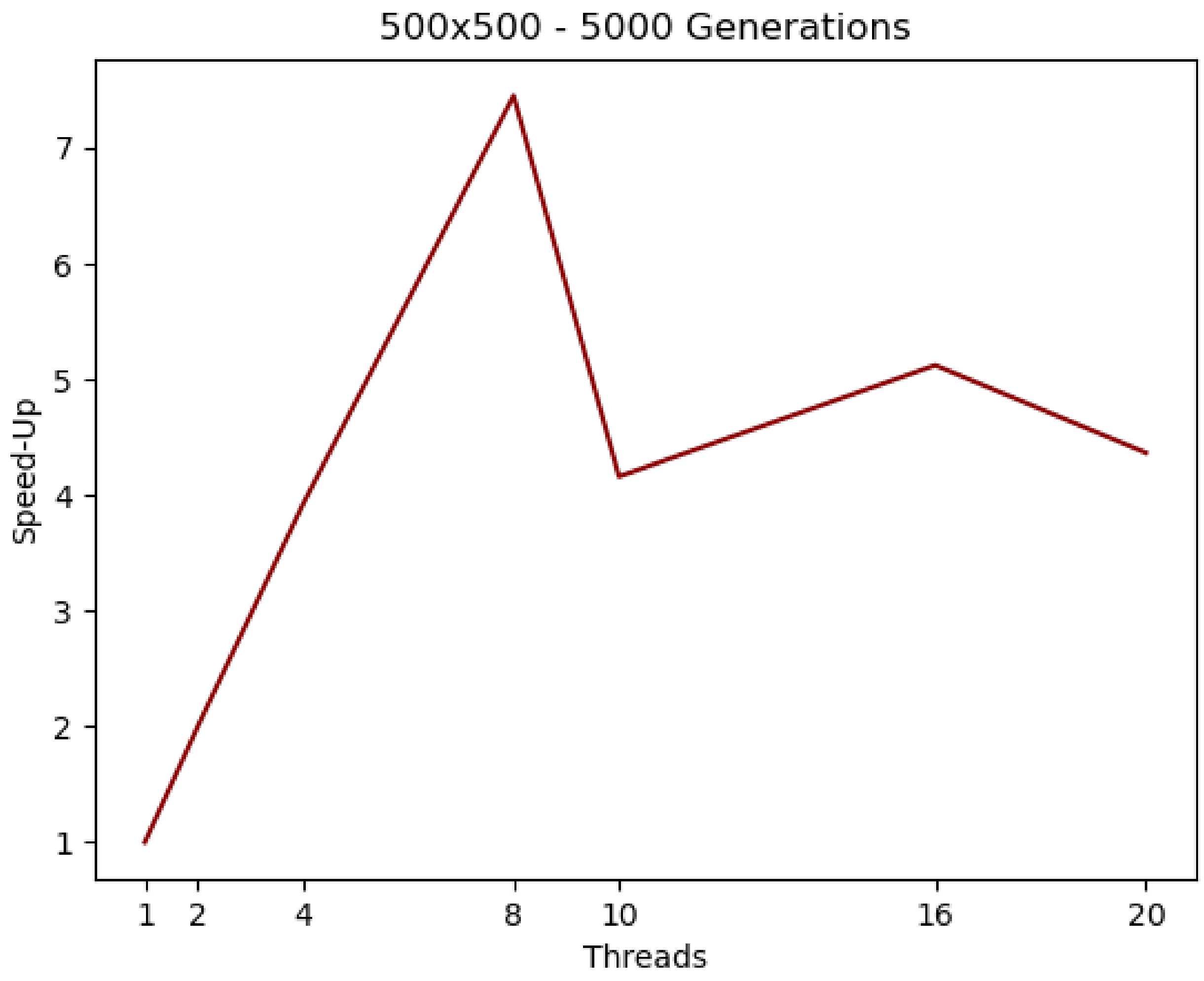

Figure 2: *This is the speed up graph for the five hundred by five hundred grid ran for five thousand generations with OpenMP. To determine each tread count's speed up, you divide the sequential time of the program by the time taken for that thread count. This is why one thread has one speed up.*

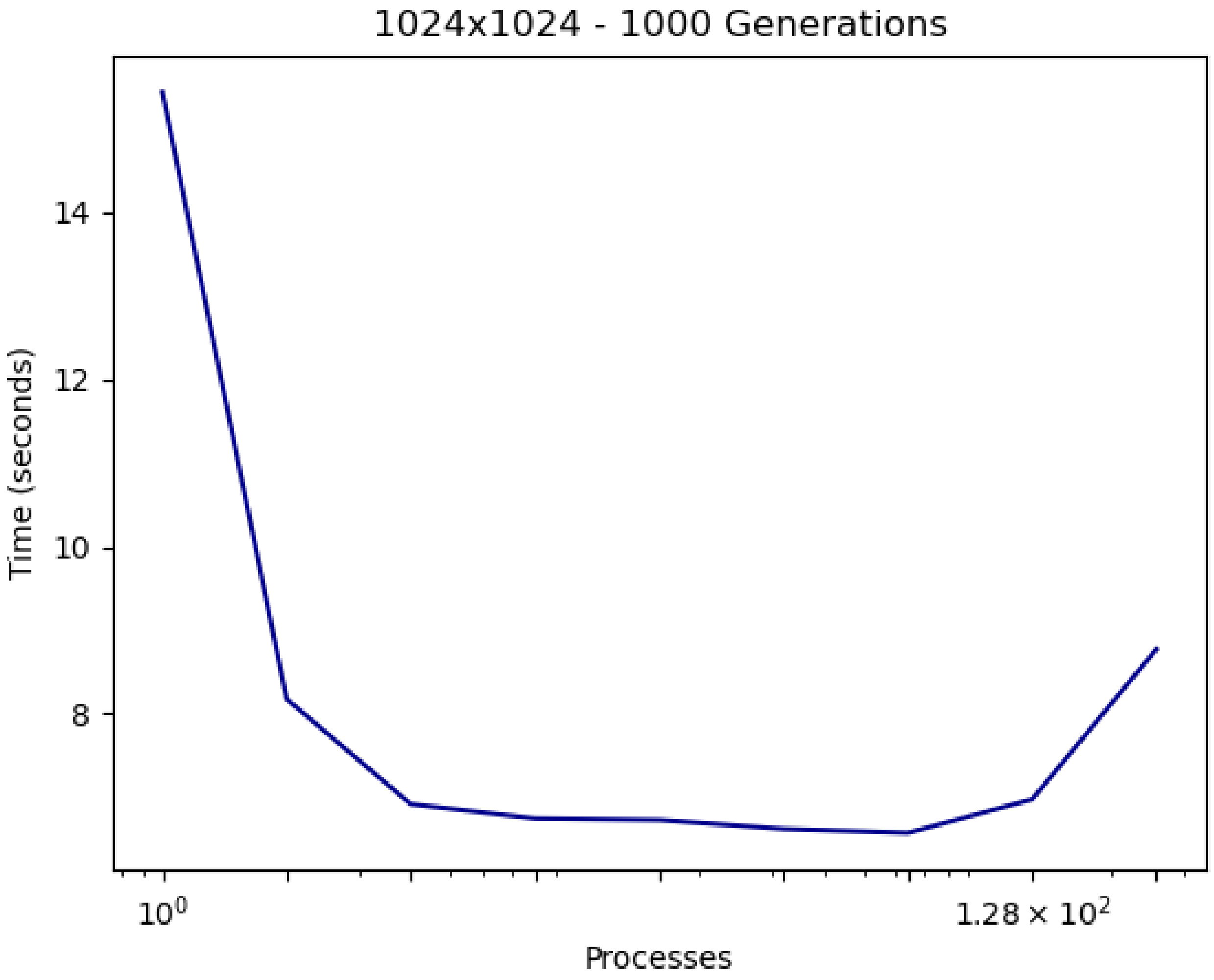

Figure 3: *This is the efficiency graph for the 1024 by 1024 grid ran for one thousand generations with OpenMPI. Each process count was ran for five iterations to determine its average value.*

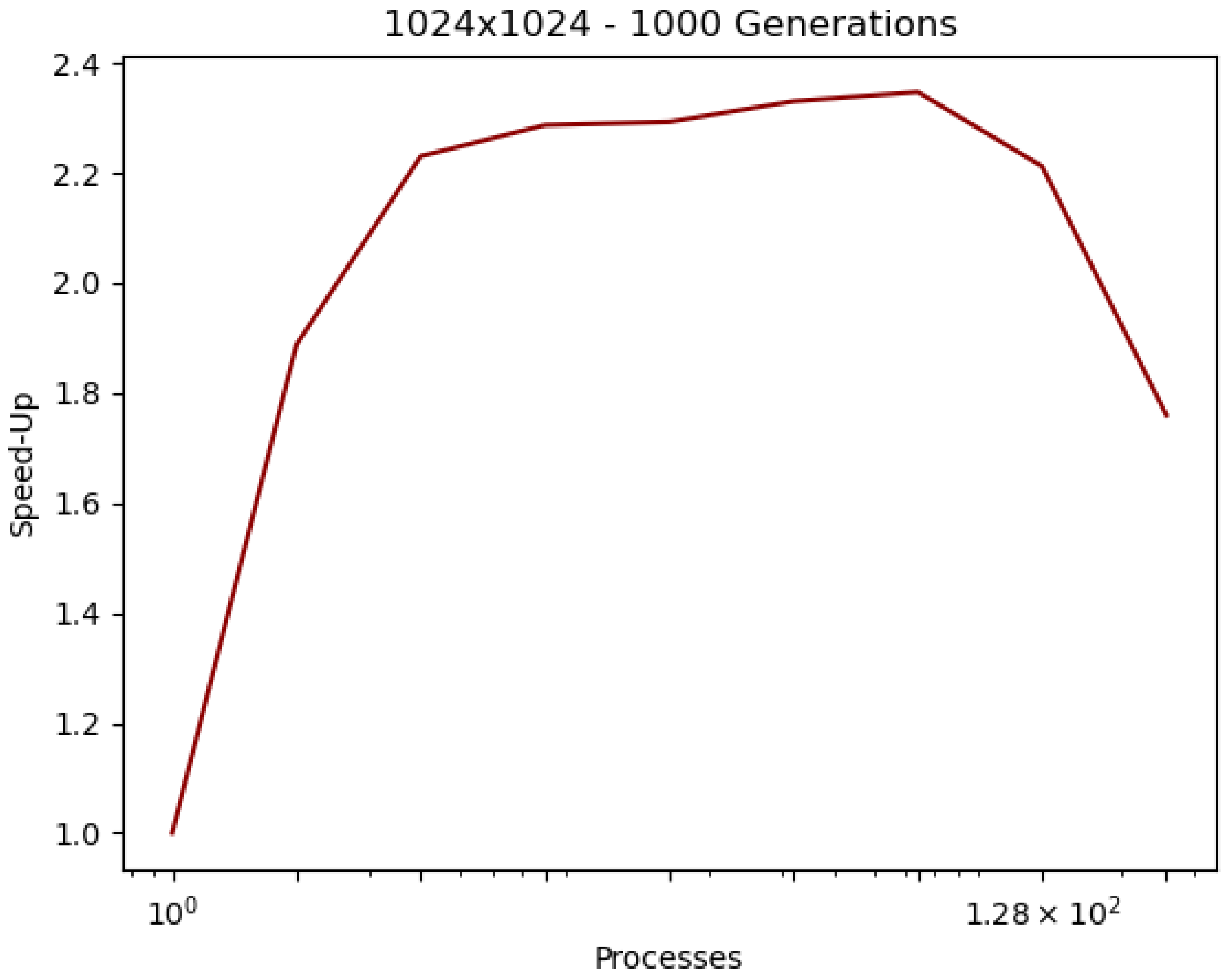

Figure 4: *This is the speed up graph for the 1024 by 1024 grid ran for one thousand generations with OpenMPI. To determine each process count's speed up, you divide the sequential time of the program by the time taken for that process count. This is why one process has one speed up.*

IV. Analysis

Figures one and two show the efficiency and speed up of the OpenMP implementation and figures three and four show the efficiency and speed up of the OpenMPI implementation. We see similar lines in the two implementations' graphs. I believe the U-shape we see associated with the OpenMPI version is due to the sharing of ghost row cells. It turns out 64 processes was the optimal amount for the OpenMPI implementation as it perfectly balanced each processes grid size with how many ghost cell shares that would need to be performed. As the process count goes beyond 64 we see a drop-off in performance. OpenMPI saw a speed up of around two at its best and OpenMP saw a speedup of nearly eight which clearly shows my OpenMP implementation was superior when compared to my OpenMPI implementation.

V. Conclusion

What I realized after this assignment is that OpenMPI as well as CUDA allows for many different avenues for solving this optimization process. Originally I attempted to gather and scatter inside of the main loop, just as you directed us not to do and it proved to cause much more harm than good. The size and shape of the grids associated with each process could vary greatly.

Overall, I believe I was successful in exploring the capabilities OpenMPI offers its users. The library enables its users to take many different approaches to solving optimization problems.

If I were to revisit this assignment in the future I would take a more test driven development approach even though debugging MPI proved to be a headache. There were some tools that enabled me to simulate each of the process as xterm windows. Even with this aid it was still incredibly difficult to locate the logic bug that was plaquing my program.