Parallel Cellular Automata

Ben Ogden
Dept. of Computer Science
University of New Mexico
Albuquerque, US
btoraptor@unm.edu

John Ringer
Dept. of Computer Science
University of New Mexico
Albuquerque, US
jrin42@unm.edu

Anthony Sharma
Dept. of Computer Science
University of New Mexico
Albuquerque, US
sharmanthony@unm.edu

Jack Wickstrom
Dept. of Computer Science
University of New Mexico
Albuquerque, US
jwickstrom@unm.edu

***Abstract*—Cellular automata have applications in a number of scientific domains, including Biology, Physics, and Computer Science [5]. Of the thousands of existing cellular automata, Conway's Game of Life (GoL) is perhaps the most famous example. Within this work we examine methods for improving the performance of GoL using parallelization with OpenMPI [2]. Parallelized versions of GoL are compared against each other and a serial implementation. Timings for each version were recorded under various configurations to measure the performance of each method. Our results indicate that parallelization can dramatically improve the run-time of GoL. However, we find there are diminishing returns when using larger and larger numbers of processes, especially when performing file I/O. We discuss these results in detail, providing analysis on bottlenecks and where potential improvements can be made. The code for this project can be found at the following GitHub repository: https://github.com/Wubbulue/CS442_Final_Project.**

I. Introduction

Cellular automata are models used to simulate complex systems. They consist of a grid of cells, with each cell each being able to take on several states. Some given rule dictates how the cells change state over time. Typically, a cell's future state is dictated by its current state and the state of its neighbors. Though simple, these rules allow for a powerful way to model complex systems. They demonstrate emergent behavior, can simulate diverse phenomena, and are interesting platforms for experimentation.

A popular automata for computational exploration is John Conway's Game of Life (GoL) [3]. Despite consisting of extremely simple states and rules, GoL is capable of universal computation as well as artistic, creative recreation. GoL consists of an infinite grid where cells can be in one of two states, often called "dead" and "alive". GoL uses the Moore neighborhood, meaning neighbors are defined by the eight cells surrounding a cell. GoL obeys the following rules at each iteration:

1) Any alive cell with fewer than 2 alive neighbors dies
2) Any alive cell with either 2 or 3 alive neighbors remains alive
3) Any alive cell with more than 3 alive neighbors dies
4) Any dead cell comes alive if it has exactly 3 alive neighbors

These rules are shown in Figure 1.

Parallelizing a cellular automaton is a useful endeavor for many reasons. First and foremost, speeding up cellular automata by splitting work up between processors allows for faster computation. Parallelization also allows for scalability; this will maintain performance with larger grids, more iterations, or more complex rules. Understanding the benefits of parallelization on an automaton can serve as a good model for extrapolating performance for large simulations generally.

II. Implementation

The serial version of Game of Life was implemented before our parallel attempts. Our implementations and the utilities they use are programmed in C++.

A. Pattern Readin

To enable the usage of interesting and useful initial state patterns, we adapted the pattern notation and loader from Golly [6]. One of the notations that Golly stores pattern data in is a dense format that specifies x and y offset, and then the specific state of each notable cell (non-default state cell). Our pattern reader parses this Golly dense notation into a boolean array, which each of the implementations can take as input.

For parallel implementations, another step is required; before the simulation can begin, the data must be dispersed among all the processes. To accomplish this quickly, we defined an MPI vector type and tactfully divided the full pattern to be distributed to each process (see section II-C for further details).

B. Serial

Similar to other serial implementations of Game of Life, we linearly traverse the given input board, checking each cell based on neighbor states. Our serial implementation essentially consists of three functions (note that for the sake of brevity some arguments are excluded, e.g. file I/O flags):

1) $get\_num\_neighbors\_alive(bool* \ A, \ int \ row, \ int \ col, ...)$
2) $evolve(bool* \ A, \ bool* \ A\_old, ...)$
3) $run(bool* \ A, \ int \ num\_iterations, ...)$

Where A represents the current grid state and A_{old} represents the grid state at the previous iteration. The $get\_num\_neighbors\_alive()$ function calculates the number of neighbors alive at cell $A[row][col]$ (where a cell is considered "alive" if it has a value of $true$). This information is used by the $evolve()$ function to update the current grid A based on the grid state at the previous iteration (A_{old}). Each cell is updated one at a time, in serial (first updating cell $A[0][0]$, then

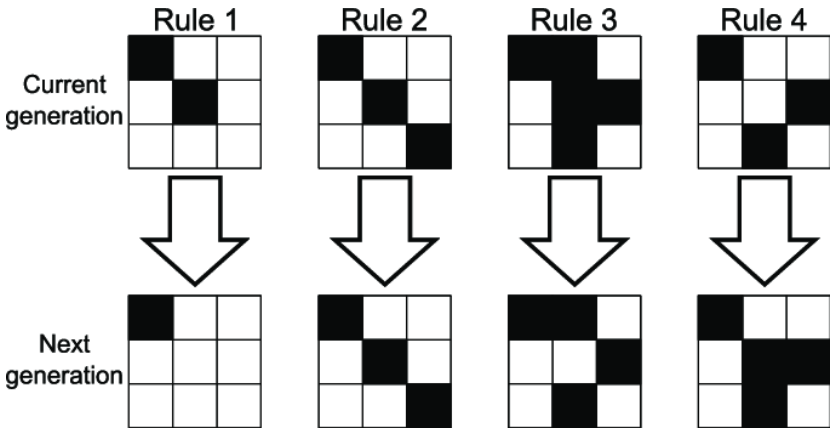

Fig. 1. Visualization of the rules of GoL through example scenarios. Alive cells are depicted in black, dead cells in white. Each respective rule refers to the middle cell in each grid. This image is taken from [4].

$A[0][1]$, and so on). The $run()$ function applies the $evolve()$ function $num\_iterations$ times to the initial state A. After each iteration, A is copied over to A_{old} using memcpy. $run()$ also handles (optional) file I/O using utility functions located in the $utils$ directory.

It is worth noting that in our implementation we use a "wrap-around" approach when considering the state of a given cell's neighbors (meaning we check 8 neighbor cells for every single cell). For example, if A is of size $M \times N$, then at cell $A[0][0]$ we consider the left neighbor to be cell $A[0][N-1]$, the up neighbor to be cell $A[M-1][0]$, and so on.

C. Naive Parallel

Our naive parallel implementation follows a similar basic structure to the serial implementation outlined above. However, rather than relying on a single process to update every single cell in the grid we now divide the grid into sub-grids to divide the work of checking neighbors and updating cell state. Assuming our grid A is of size $N \times N$ and we have p processes, each process is assigned to a sub-grid of size $\frac{N}{p} \times \frac{N}{p}$ [1]. Each process can then carry out the GoL rules on their own sub-grid, with the important caveat that bordering process grids (sub-grids where the perimeter cells of each meet) need to communicate state information with one another to ensure the correctness of the simulation.

The implementation takes advantage of the fact that for updating a given cell $A[i][j]$, we only need to read the state of cells within the Moore neighborhood (left/right, up/down, and the four diagonals) to calculate the next state of $A[i][j]$. Thus, for cells that are not on the perimeter of the sub-grid the relevant process only needs to read from and write to its own sub-grid. However, cells on the perimeter of sub-grids need to receive information about neighbors which belong to a different processes' sub-grid. Thus, the Naive Parallel implementation requires some communication in order to carry out the simulation correctly. For communication we use $MPI\_ISend()$ and $MPI\_IRecv()$ between appropriate processes, with buffers containing state information for relevant cells. Figure 2 provides an illustration of how sub-grids are divided as well as communication requirements for different cells in a small example grid.

[1]The current code assumes A is a square grid and that p evenly divides N

D. Improved Parallel

As with the Naive Parallel implementation, the Improved Parallel implementation follows a similar basic structure as outlined in section II-B. Given that the communication overhead involved in the Naive Parallel implementation is caused solely by perimeter cells, we believed a potential improvement over the Naive Parallel implementation would be to use shared memory windows in these regions. Thus, a shared memory version of the Naive Parallel version described above was implemented in an attempt to speedup the parallelized GoL by reducing communication costs. Functionally the Improved Parallel version is very similar to the Naive Parallel version described above, with the only difference being that rather than using $MPI\_ISend()$ and $MPI\_IRecv()$ to communicate information about perimeter cells, a shared memory window is established for each perimeter region using $MPI\_Win\_allocate\_shared()$ so that all processes may access information about the perimeter cells of any sub-grid.

E. Visualization

To support the visualization of simulations, we implemented file I/O utilities so that that the world grid could be saved to a binary file at the end of each iteration. A corresponding renderer using OpenCV [1] was implemented to display a simulation's binary file as a video. The renderer processes the simulation such that every iteration in the original simulation corresponds to a frame in the rendered video.[2] An example

[2]We note that the renderer struggles to display extremely large grid sizes due to conflicts between the chaotic, noise-like appearance of GoL simulations and the nature of video compression. However, the renderer performs well for smaller grid sizes.

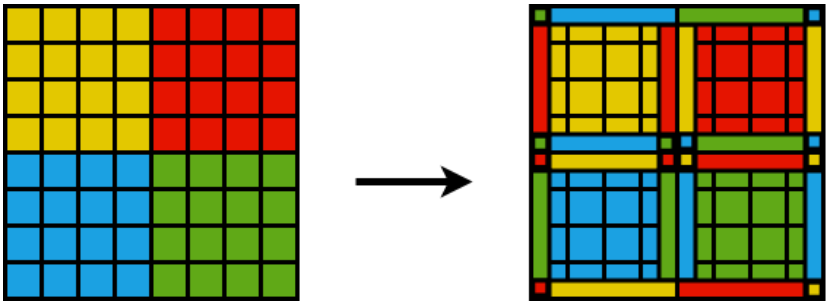

Fig. 2. (Left) Figure showing a 16x16 grid divided among 4 processes. Each sub-grid is colored differently. (Right) Communication requirements for each cell in the Naive Parallel implementation. If a color is present in a given cell, then that cell must receive information from the process corresponding to the color.

simulation is shown on the main page of the GitHub repository, or one can view an example rendering here.

III. Results

Our group decided to divide the tests into two separate parts. We decided to analyze the performance of our Serial, Naive Parallel, and Improved Parallel versions both with and without file I/O. We wanted to first analyze each method without file I/O to guarantee that we were only correlating observations to differences in implementation method and number of processes used, as the cost of file I/O can dominate the run-time and make it more difficult to compare methods on the basis of pure computation. All tests were run on the Wheeler supercomputer at UNM's Center for Advanced Research Computing.

A. Pure computation

Our group agreed to test using the following data set. We tested on board sizes with 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096, and 8192×8192 total cells. We ran tests on the Serial version, allocating one node. We ran tests on the Naive Parallel version with 4,16, and 64 processes, with a maximum of 8 processes per node (8 processes is the maximum number of processes one can allocate per node on the Wheeler supercomputer). The Improved Parallel version was tested only with 4 processes due to the constraints on the number of processes/node on the Wheeler supercomputer (with our implementation shared memory only works if processes are on the same node). All tests were given the same starting state.

Looking at Figure 3 we see that as the dimension size of the board increases, the discrepancy between our serial and parallelized implementations increases and runs with higher process counts complete faster in comparison to runs with lower process counts. For example, with board dimensions 1024 and higher, process count 64 completes first, process count 16 second, process count 4 next, and our serial version comes in last. One surprising detail of Figure 3 is that for board dimension 512×512, a process count of 64 had a smaller completion time, compared to process counts 4 and our serial implementation. This was not expected as in most cases, with small input data, using more processes had worse performance. This is usually caused by the cost of communication out weighing the benefits of using many processes to complete computations, which with small data time saved is minimal. However this experiment defies that assumption.

Another result from Figure 3 worth noting is the completion times for 4 processes on our shared memory MPI implementation (Improved Parallel). While expected to have a faster run time than its 4 process MPI naive counterpart, both managed to run with similar times for all board dimension sizes tested on. The reason for this occurrence is explained in section IV.

B. With I/O

For many practical applications, data needs to be recorded during a run. When including file read-in at the beginning of a GoL sequence and writing state to a file each iteration, runtimes vary highly. Similar to the non-I/O testing, we tested on board sizes with 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096, and 8192×8192. All tests were run on Wheeler and were given the same starting state.

Figure 4 shows that parallelized versions still perform better than the serial version. However, when writing to a file, parallel versions all perform nearly the same - more processes offer negligible performance gains. As with the non-I/O testing, the shared memory implementation performs as well as the 4-process Naive Parallel implementation.

C. Efficiency

After collecting all the performance data, we standardized the results to better understand the true productivity of each different method. Each run time was scaled to factor in the size of the board it was given and the number of available processes for computation.

Both Figure 5 and Figure 6 show that, as expected, large process counts have poorer performance on smaller board sizes. The figures differ in one major facet - optimal performance at large board sizes. Figure 5 reveals that at large board sizes, all process counts and implementations perform nearly the same amount of computation per process. Contrarily, when I/O is involved this is not true. Figure 6 shows that runs with fewer processes are less efficient at large board sizes.

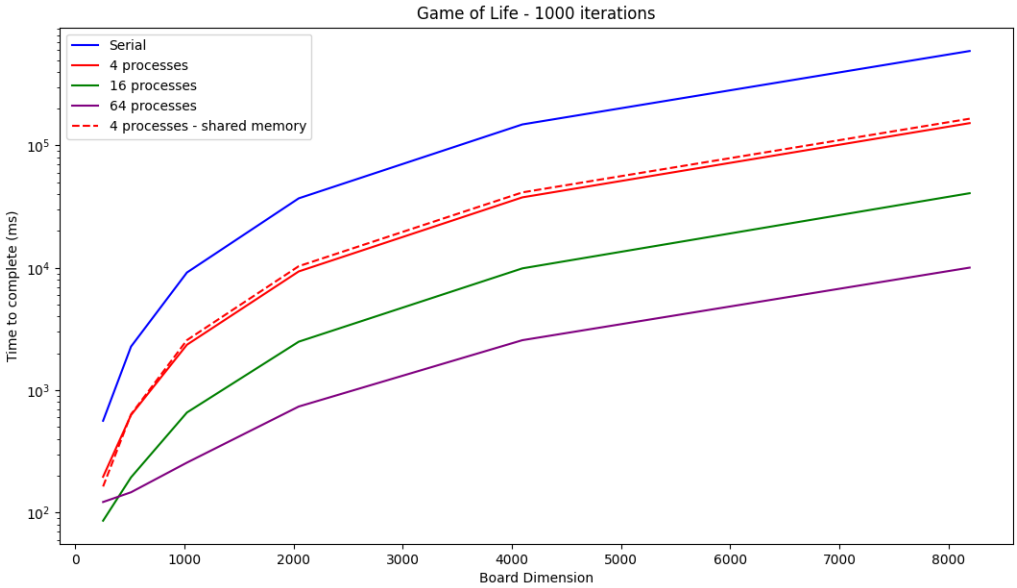

Fig. 3. Time to run 1000 iterations of Game of Life with various board sizes and implementations

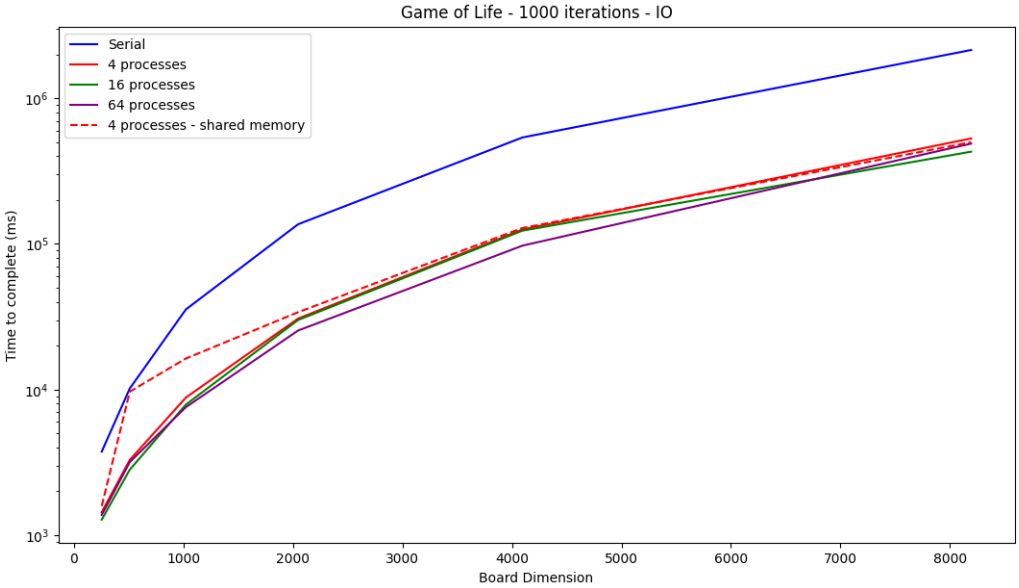

Fig. 4. Time to run 1000 iterations of Game of Life with various board sizes and implementations. An initial board state is read from file and every iteration, the state is stored to a playback file.

IV. Discussion and Conclusions

As expected, adding I/O substantially decreases the overall speed of execution. I/O active implementations add more communication and time-consuming file system operations to each GoL iteration. While all implementations, including serial, must perform I/O, parallel implementations add even more overhead. We chose to gather data on a single process to perform I/O in one single action. There were other possible approaches; having each process write to its output file was one that was particularly intriguing. However, to stay consistent with the serial implementation, we wanted all output files to be conglomerated data - even if that meant that parallel

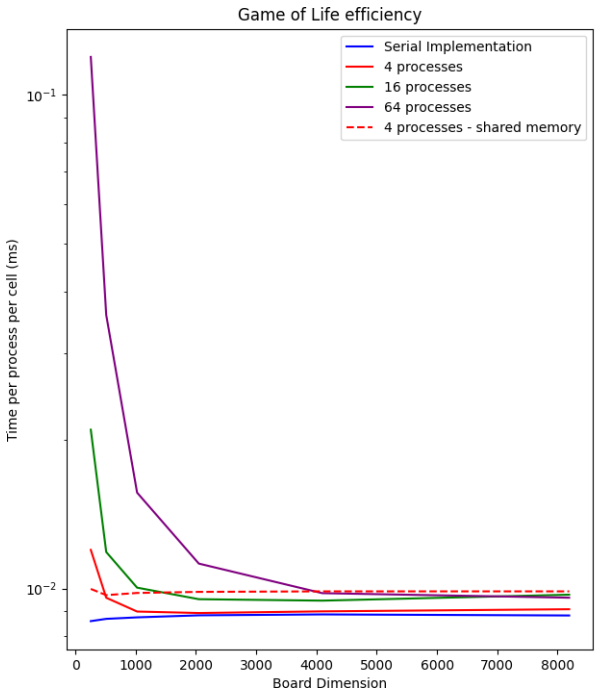

Fig. 5. Average time for one process to calculate one cell of Game of Life.

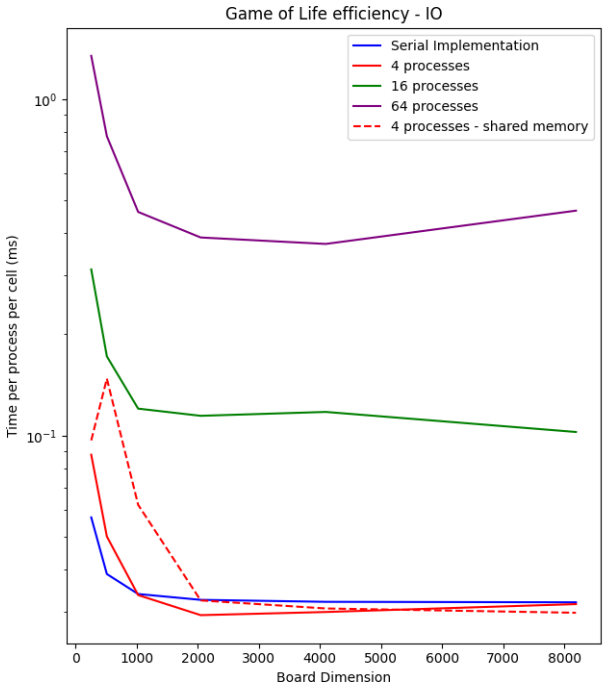

Fig. 6. Average time for one process to calculate one cell of Game of Life with I/O.

implementations had extra work to do.

Figure 6 shows the consequences of gathering data. As the process count increases, getting all data into one process becomes harder. Network contention in particular becomes a painful roadblock to performing quick gather operations. Thus, there are a few possible approaches to handling I/O better. If memory permits it, gathering many iterations and writing in bulk could be a good solution. As mentioned before, having each process write its own log could also be beneficial.

Another method hypothesized to assist with I/O was using shared memory. The motivation behind shared memory is that it allows the processes to avoid communicating to gather the data; instead, they just need to indicate to a root process that they have completed the current iteration. This would hopefully save a good amount of time every iteration, adding up to substantial performance gains. Unfortunately, Figure 4 displays a non-existent improvement. After deliberation, we have concluded that this lack of performance gains was caused by the two costly calls of $MPI\_BARRIER$ that our implementation demanded. This suggests that the savings for avoiding sending data are not significant; the cost of synchronization between all processes that must occur every iteration is far more expensive.

Whether it is an issue with poor implementation or a property of shared memory itself, this is one of the topics where further investigation is needed.

V. Contributions

All team members contributed to this project. Jack W. implemented both parallelized versions of GoL as well as the renderer. Anthony S. implemented reading in Golly patterns and also on generating results/figures. Ben O. worked on various code utilities (e.g., file I/O) and generating results/figures. Jack R. implemented the serial version of GoL and various code utilities. All team members contributed to discussion and feedback on each other's ideas throughout the course of the project.

VI. Acknowledgement

We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the high performance computing resources used in this work.

References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[2] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[3] Mathematical Games. The fantastic combinations of john conway's new solitaire game "life" by martin gardner. *Scientific American*, 223:120–123, 1970.
[4] Takayuki Hirose and Tetsuo Sawaragi. Extended fram model based on cellular automaton to clarify complexity of socio-technical systems and improve their safety. *Safety Science*, 123:104556, 03 2020.
[5] K. Preston and M.J.B. Duff. *Modern Cellular Automata: Theory and Applications*. Advanced Applications in Pattern Recognition. Springer US, 2013.
[6] Andrew Trevorrow and Tom Rokicki. Golly game of life home page, Aug 2022.