C2: Advance Research Computing

Coursework Assignment

University of Cambridge

Department of Physics

Prepared by:
Sabahattin Mert Daloglu

Word Count: 2706

March 28, 2024

1 Introduction

The Game of Life, a cellular automaton devised by Cambridge mathematician John Horton Conway, serves as a captivating example of how simple rules can orchestrate complex dynamics within a grid-based universe. In this game, every cell on a two-dimensional grid can exist in one of two states: alive (1) or dead (0). The fate of these cells across successive generations is determined by their initial states and the states of their eight neighbors, governed by four rules of the game[3]:

- A live cell with fewer than two live neighbors dies, as if by underpopulation.
- A live cell with two or three live neighbors lives on to the next generation.
- A live cell with more than three live neighbors dies, as if by overpopulation.
- A dead cell with exactly three live neighbors becomes alive, as if by reproduction.

The initial configuration of the grid and the number of iterations—or updates—are set by the user, setting the stage for various patterns to emerge. These patterns range from stable entities (still lives) that maintain their structure indefinitely, to oscillators that cycle through a series of states, and even spaceships, which traverse the grid over time. Beyond their aesthetic appeal, these patterns serve as critical tests for the algorithm's accuracy in this study.

2 Selection of Solution Algorithm and Prototyping

The journey to selecting an efficient algorithm for the Game of Life began with exploring various approaches and culminating in the adoption of the most rapid solution. The process was kick-started with a brainstorming phase, where a mind map guided the initial prototyping efforts. The primary strategy examined was a neighbor-counting algorithm that utilized nested loops to traverse the grid and calculate the live neighbors for each cell. This approach, coupled with a subsequent function to update the grid based on these counts and apply the game's rules, represented the most intuitive—but ultimately the least efficient—method. Initial experiments were performed with this approach along with profiling. The neighbor counting algorithm was realized to be not efficient due to its memory access pattern which is not contiguous when counting neighbors at different rows. A more efficient and easy-to-optimize algorithm was chosen to be a 2D convolution for counting the live neighbors. The kernel of choice for this approach was a 3x3 kernel filled with ones, excluding the central element to exclude the cell itself from its neighbor count. However, it was noticed that this kernel is not separable into two 1D kernels which could potentially reduce computational load by decreasing the number of necessary multiplications. To utilize the advantages of separable kernels, the initial 3x3 kernel can be used with all elements as 1, making it separable. By employing separable 3x1 and 1x3 kernels, the multiplication operations reduce from nine to six, enhancing the algorithm's efficiency [2]. After the convolutions, we can account for the central value by subtracting it from the convoluted result or adjusting the rules of the game to account for this change. Figure 1 illustrates the workflow of the separable convolution method proposed.

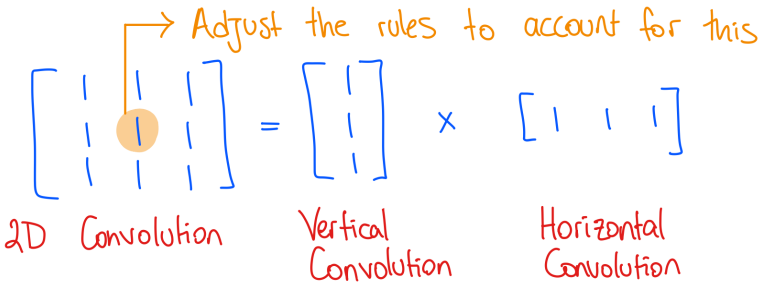

Figure 1: A workflow of the proposed separable convolution. A 2D convolution kernel can be separated into two 1D convolution kernels, and later the central value can be subtracted from the convoluted result.

3 Development, Experimentation, Profiling, and Optimization

3.1 Experimentation and Profiling

The initial ideas for applying the game rules were tested by experimenting with a neighbor counting algorithm consisting of a nested for-loop that iterated over each grid element. This accompanied by a function that updates the grid based on the output from the neighbor counting algorithm is the naive approach to the Game of Life. Firstly, a *Grid* class is created for initializing a dynamically allocated array of integers to represent the grid of the simulation. This array is one-dimensional, but an indexing operator is also defined so grid elements can be accessed in 2D convention. The functions that will be applied to the grid are defined as the method function of the class *Grid*. **countLiveNeighbors** function shown in Algorithm 1, and **updateGrid** function shown in Algorithm 2, are defined in the **grid** library file. **countLiveNeighbors** iterates over each element and checks all 8 neighbors by summing their values. This function is designed for implementing periodic boundary conditions in which the grid wraps around in all 4 directions. It is later passed into the **updateGrid** function which applies the four-game of life rules based on the number of living neighbors. It is important to note the absence of if-statements in the **updateGrid** function. Initially, the game rules were implemented by checking the current state of the cell and the number of alive neighbors with 4 if-statements corresponding to the 4 rules of the game. In this way, the time taken for updating a grid of size $1,000x1,000$ over 10-time steps was calculated to be 1833.035 ms on average with repeated measurements. Later, the Algorithm 2 was implemented with the branch prediction optimization which reduces the logical checks by replacing if-statements with a ternary operator. The time taken for updating a grid of the same size over 10 time steps was calculated to last 74.169 ms on averaged repeated measurements. This speed-up made sense since the number of branching points provided to the compiler is reduced hence making the code more predictable for the CPU's branch predictor by slightly reducing the instruction count. Although this naive approach was not used for the final algorithm, the branch optimization for applying game rules was carried out to the separable-convolution algorithm described in the next section.

Algorithm 1 Count Live Neighbors with Periodic Boundary Conditions

```
function CountLiveNeighbors(row, col)
    live_neighbors ← 0                                   ▷ Initialize live neighbors count
    for dx ← −1 to 1 do                                  ▷ Iterate over the row offset
        for dy ← −1 to 1 do                              ▷ Iterate over the column offset
            if dx = 0 & dy = 0 then
                continue                                 ▷ Skip the cell itself
            end if
            x_neighbour ← (row + dx + size1) mod size1   ▷ Wrap row index with modulo
            y_neighbour ← (col + dy + size2) mod size2   ▷ Wrap column index with modulo
            live_neighbors ← live_neighbors + grid[x_neighbour, y_neighbour]   ▷ Accumulate live neighbors
        end for
    end for
    return live_neighbors                                ▷ Return the count of live neighbors
end function
```

The code for this initial approach was profiled by using XCode's **Instruments** analyzer with the **Time Profiler** template. In preparation for profiling the code is compiled with a debugging flag and optimization of -O2. In general, when profiling and debugging, compiler optimization is not recommended to observe the original structure of the code without the intervention of the compiler optimization. However, for practical purposes and to understand the real-world performance of the methods, -O2 optimization is preferred in this study. The algorithm is run for updating a grid of size $1,000x1,000$ over 100-time steps with the profiling results shown in Figure 2.

Around 70% of the run time was found to be occupied by the **updateGrid** function whereas the remaining 30% by the **countLiveNeighbors** function. This is expected since both functions iterate

Algorithm 2 Update Grid by Applying Game of Life Rules

```
function UpdateGrid
    new_grid ← new int[size1 × size2]                          ▷ Allocate new grid
    for i ← 0 to size1 − 1 do                                  ▷ Iterate over rows
        for j ← 0 to size2 − 1 do                              ▷ Iterate over columns
            live_neighbors ← CountLiveNeighbors(i, j)
            index ← i × size2 + j                              ▷ Map 2D indices to 1D
            alive ← grid[index] == 1
            survive ← alive and (live_neighbors == 2 or live_neighbors == 3)
            reproduce ← ¬alive and live_neighbors == 3
            new_grid[index] ← survive or reproduce?1 : 0
        end for
    end for
    swap(grid, new_grid)                                       ▷ Swap old grid with new grid efficiently
    delete[] new_grid                                          ▷ Deallocate the swapped old grid
end function
```

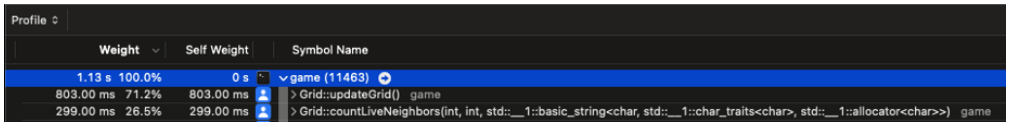

Figure 2: Time profiler output from the **Instruments** analyzer showing the run time distribution of the algorithm into its functions.

over the whole grid but the **updateGrid** function also initializes and allocates a new grid for the updated grid.

While the neighbor counting algorithm offers a direct approach, its suitability for domain decomposition—especially in scenarios aiming to overlap communication with computation—is limited. Consequently, a more refined convolution method, which incorporates kernel separation alongside domain decomposition, serves as a better alternative. This technique, when used in conjunction with the Message Passing Interface (MPI), significantly enhances efficiency and performance. The integration of kernel separation with domain decomposition and MPI is explored in greater detail in the Optimization section.

3.2 Development and Optimization

The development and optimization phase is marked by the strategic use of parallelization to enhance the computation speed of Conway's Game of Life grid evolution. Domain decomposition plays a pivotal role in this process, segmenting the larger grid into smaller, manageable sections (local grids) allocated across different MPI ranks. Each rank, representing a core on the HPC node, executes a fraction of the simulation, necessitating effective communication to synchronize border cell data through halo exchange.

The choice of a 2-D over a 1-D domain decomposition is driven by efficiency; it minimizes data transfer volume to $\alpha \times \sqrt{\text{nranks}}$, compared to $\alpha \times$ nranks in 1-D decomposition. Cartesian communicators further refine this process by aligning the number of processes with the grid's dimensional constraints, optimizing communication paths, and reducing overhead. However, this approach introduces complexity, particularly in managing communications across cell corners, involving up to eight interactions per cell.

To address these challenges and streamline communication, a separable convolution algorithm is introduced. By dividing the convolution into vertical and horizontal components, the algorithm significantly reduces the necessity for corner communications. This approach simplifies the 2-D domain decomposition and effectively shrinks the communication overhead of the algorithm.

Implementing rectangular domain decomposition accommodates grids that do not neatly divide across the processor grid, which is expected given arbitrary MPI rank inputs. The **MPI_Scatterv** function is crucial here, allowing for the distribution of unevenly sized local grids across processes. This

flexibility is essential for robustly handling various grid and rank configurations but on the downside introduces the potential bottleneck of ranks with disproportionately large local grids.

Each process receives the main code in Algorithm 3, which separates the convolution into vertical and horizontal sections. The **AddVerticalPadding**, pads the local grid by adding one halo row above and one halo row below to store the up and down neighbors' boundary cells. Then, **VerticalHaloExchange** function is applied to the grid which uses **MPI_cartesian** to exchange the boundary cells with the top and bottom neighbors as shown in 4. Non-blocking **MPI_Isend** and **MPI_Irecv** functions are used during the halo exchange so that while sending the bottom row to the top neighbor, the top row from the bottom neighbor is received. In this way, the communications for send and receive are overlapped thus preventing deadlocks and ensuring smooth data exchange among neighboring ranks. A similar simultaneous communication is performed by sending the bottom row to the bottom neighbor while receiving the bottom from the top neighbor. After the vertical halo exchange, vertical convolution is applied by **VerticalConv** function. This function applies a 3x1 convolution to the grid of the original size before vertical padding. The vertically padded halos are used only for calculating the convolution and after the convolution the grid size is reduced by 2 rows back to its original size. The sixth line of the Algorithm 3 shows the blocking function **MPI_Barrier** to wait for each process vertical communication.

Algorithm 3 Parallel Game of Life Execution

```
1:  for time = 0 to time_steps do
2:      conv_grid.setGrid(local_grid.getGrid())      ▷ Deep copy local grid to convolution grid
3:      conv_grid.AddVerticalPadding()                ▷ Add vertical padding
4:      conv_grid.VerticalHaloExchange()              ▷ Exchange vertical halo cells
5:      conv_grid.VerticalConv()                      ▷ Perform vertical convolution
6:      MPI_Barrier()                                 ▷ Sync after vertical communication
7:      conv_grid.AddHorizontalPadding()              ▷ Add horizontal padding
8:      conv_grid.HorizontalHaloExchange()            ▷ Exchange horizontal halo cells
9:      conv_grid.HorizontalConv()                    ▷ Perform horizontal convolution
10:     MPI_Barrier()                                 ▷ Sync after horizontal communication
11:     local_grid.ApplyGameRules(conv_grid)          ▷ Apply rules to local grid
12:     MPI_Barrier(cart_comm)                        ▷ Final sync after rule application
13: end for
```

Algorithm 4 Vertical Halo Exchange with MPI_Isend and MPI_Irecv

```
1:  procedure VerticalHaloExchange(rank, ranks, cart_comm)
2:      Determine the ranks of up and down neighbors.
3:      up, down ← MPI_Cart_shift(cart_comm, 0, 1)
4:
5:      // Exchange boundary cells with neighboring processes
6:      Send the top row to the up neighbor.
7:      MPI_Isend(top row, to up neighbor)
8:
9:      Receive the top boundary row from the down neighbor.
10:     MPI_Irecv(top boundary, from down neighbor)
11:
12:     Send the bottom row to the down neighbor.
13:     MPI_Isend(bottom row, to down neighbor)
14:
15:     Receive the bottom boundary row from the up neighbor.
16:     MPI_Irecv(bottom boundary, from up neighbor)
17: end procedure
```

After performing the vertical communication and convolution, the same calculations are performed in the horizontal direction for a 1x3 kernel size. Finally, the updated rules of the game of life, which

accounts for the inclusion of the cell itself in the convolution, are applied in line 11 of the Algorithm 3. It is important to notice that by separating the convolution and communication, the corner neighbors of each local grid are automatically communicated without explicit MPI implementation. In order to understand how this is possible let's consider the top corner boundary cell. After the first vertical communication, this corner neighbor is passed down to the right neighboring local grid. The vertical convolution convolves the corner value into the top row of the right neighboring grid. Finally, during horizontal halo exchange and convolution, the previously right-top corner neighbor element is now convolved into the top right cell of the grid. Overall, the implementation of separable convolution kernels with 2D Halo domain decomposition removes the need for corner boundary communication, simplifying the algorithm. Nevertheless, a disadvantage of this method is that the scattering of local grids to processes is managed by reorganizing the full grid such that each local grid is contiguous in memory, and ready for **MPI_Scatterv** function. This is made possible by implementing **reogranizeGrid** function which is called before scattering and the inverse of this function **InverseReogranizeGrid** to reverse this layout back to its original after gathering the local grids into the main large grid.

The effectiveness of this parallelized implementation is empirically validated through simulations across varying grid sizes—$5,000x5,000$, $10,000x10,000$, and $20,000x20,000$—over 100-time steps for a range of MPI processes. These experiments are complemented by theoretical speed-up predictions based on Amdahl's Law[1], which underscores the impact of parallelizable algorithm portions on overall computational efficiency.

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \tag{1}$$

The $S(N)$ term is the speed-up achieved using N processes. For this study, a parallelizable proportion, p, of 0.95 is adopted, aligning theoretical speed-ups closely with observed experimental outcomes. This dual approach not only demonstrates the algorithm's scalability but also its real-world applicability to large-scale simulations, as depicted in the run time versus MPI ranks graph.

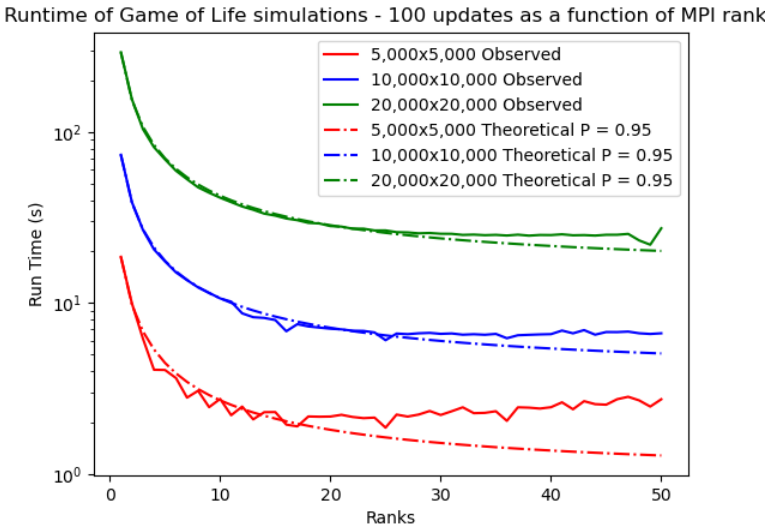

Figure 3: Run time of Game of Life simulations for 100 updates as a function of MPI processes.

For all three grid sizes, the time reduction with the increased size of ranks gets saturated around 20 ranks. This value indicates that after 20 ranks communication overhead of the increased number of processes limit the speed-up gained by parallelization. A bottleneck might exist in the algorithm assuming not all portions of the program are parallelizable (0.05), which sets a natural limit to the speed-up. Nevertheless, the plateaued values of the run times are comparable to the theoretical values calculated according to Equation 1 with a p-value of 0.95.

The above analysis is performed for single-thread simulations. Effective thread parallelization using OpenMP is also of significant importance for utilizing shared memory parallelization. OpenMP enables

the utilization of thread communication within the same process to communicate and share data more efficiently as it eliminates the need to send messages among threads. This can reduce communication overhead. One important use case of OpenMP in this coursework is to parallelize the nested for-loops in the **VerticalConv** and **HorizontalConv** functions. OpenMP's **#pragma omp parallel for collapse(2)** directive distributes the loop iterations across available threads by collapsing nested loops into single parallel loops. The choice of which nested for-loop to parallelize using OpenMP is critical since the computation between loops should be independent. There is a trade-off between the use of cores for MPI processes and OpenMP threads on a single computer node. The product of the number of OpenMP threads per MPI process and the number of MPI processes should not exceed the total number of cores on a node in CSD3, which is 76 cores. Hence, we explore different combinations of both inter-node parallelism with MPI and intra-node parallelism with OpenMP threads. The goal is to identify the configuration that maximizes the performance of the Game of Life code. Table 1 shows different combinations of MPI processes and OpenMP threads for a grid size of $5,000$x$5,000$ over 100 updates. The most optimal result was found to be 19 MPI processes with 4 OpenMP threads per rank. This balance is logical since using OpenMP to handle parallelism within an MPI rank reduces the number of MPI processes needed for the same amount of speed-up. Moreover, the saturation of speed-up around 20 ranks, as observed in Figure 3, agrees with this optimal combination of MPI and OpenMP.

Table 1: Game of Life simulation for a grid size of $5,000$x$5,000$ over 100 updates.

MPI Processes	OpenMP Threads	Time (seconds)
1	76	19.59
2	38	10.87
4	19	4.93
19	4	2.97
38	2	3.24
76	1	3.05

4 Summary

This report presents an optimized and parallelized implementation of Conway's Game of Life algorithm. Initially, various approaches were prototyped, with a focus on improving the naive neighbor counting method through a separable convolution technique, significantly enhancing performance by reducing computational complexity. Profiling and experimentation identified bottlenecks in memory access patterns and led to the adoption of a domain decomposition strategy, employing MPI for distributed computing and optimizing communication through halo exchange and Cartesian communicators. A critical method was the application of separable convolutions, which minimized MPI communications and simplified the algorithm by removing the need for explicit corner cell communication. Finally, experimentation was conducted to find the optimal balance between MPI processes and OpenMP threads within a single HPC node environment, revealing an ideal configuration that maximizes computational efficiency. This configuration showcased the effectiveness of leveraging both MPI and OpenMP for maximizing computational efficiency, with experiments demonstrating significant speed-ups and highlighting the limitations imposed by communication overheads beyond certain MPI process counts. The findings underscore the importance of carefully selecting parallelization strategies to enhance the performance of complex simulations on modern computing architectures.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[2] C.-F. Wang. A basic introduction to separable convolutions. *Towards Data Science*, Aug 2018. [Online; accessed 12-March-2024].

[3] Wikipedia. Conway's game of life — wikipedia, the free encyclopedia, 2023. [Online; accessed 12-March-2024].

A Appendix: Generative AI

ChatGPT 3.5 was used for suggesting alternative wordings, grammar suggestions, and proofreading while writing the report. The following prompts were used during this process

- "How can I write this equation in LaTeX format? *[input equation]*"
- "Provide BibTex citation format for this website textit[input website]"
- "What is the rule of 5 in C++ and how important it is in class definitions?"
- "What is another word for *[input word]*?"
- "Is this sentence grammatically correct? *[input sentence]*"
- "Is this paragraph clear for a reader? *[input paragraph]*"
- "How to rephrase this sentence to make it more clear? *[input sentence]*"

For the report writing, the outputs were partially adapted in a way that only alternative wordings were used and not the whole output while rephrasing the Summary and Introduction parts of the report. The LaTex code for Equation 1 was adapted from the suggestion. The suggestions for BibTeX citations are also adapted.

Furthermore, the suggestions from the autocomplete feature of 'GitHub Copilot' were utilized during the documentation of the software, and code development of the project in below areas:

- Test functions for the Grid class methods contained in the **test_grid.cpp** were developed with the help of Copilot suggestions, mainly the example grids were adapted.
- Python scripts **mpi_openmp.py** and **run_ranks.py** to generate run-time analysis data used in the parallelization section was developed using the suggestions from Copilot when implementing the subprocess call.
- Indexing and logic used in the **reorganizeGrid** and **inverseReorganizeGrid** functions were developed by adapting suggestions from Copilot.