

Parallel Implementation of Conway's Game of Life

Micaela Del Longo, Federico Williamson

Universidad Nacional de Cuyo

Abstract: This work proposes a parallel implementation of Conway's Game of Life, a classic cellular automaton, using the Message Passing Interface (MPI). This approach aims to exploit the inherent parallelism of the game to efficiently distribute computation across multiple processes, thereby harnessing the power of modern parallel computing architectures.

Keywords: Conway's Game of Life, MPI, Domain decomposition, Master-Worker algorithm.

Contents

1. Introduction	2
2. Game of Life	3
3. Sequential Pseudocode	4
4. Parallelisation Analysis	5
4.1. Decomposition Strategy	5
4.1.1. Grid Domain Decomposition	6
4.1.2. Horizontally Striped Domain Decomposition	6
4.1.3. Vertically Striped Domain Decomposition	7
4.2. Parallel Algorithm Model	7
4.3. Communication Model	8
4.4. Other Optimisation Opportunities	8
5. Parallel Design and Implementation	9
5.1. Design	9
6. Experiment Design	10
6.1. Parameters	10
6.1.1. Sequential Implementation Parameters	10
6.1.2. Parallel Implementation Parameters	10
6.1.3. Common Parameters	10
6.2. Metrics	11
6.3. Scalability Metrics	12
6.4. Objectives	12
7. References	12
I. Appendix	13
I.I. Initial State Representation	13
I.II. Build and Execution	13

1. Introduction

Conway's Game of Life (Gardner, 1970) is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, where its evolution is determined by its initial state, needing no further input from a player. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

Game of Life's applicability spans diverse fields, including biology, physics, computer science, and beyond. Studying the dynamics of complex systems through it offers insights into pattern formation, self-organisation, and emergent phenomena, with potential implications for designing efficient algorithms, modelling biological processes, and simulating natural systems. Thus, the importance of understanding and efficiently simulating Conway's Game of Life extends beyond recreational curiosity.

Regarding parallel implementations of Game of Life, in some (Panagiotopoulos, 2019; Papa-georgiou, 2020) domain decomposition is a common strategy employed to divide the computational workload among multiple processing units. This approach involves breaking the grid representing the cellular automaton into smaller subdomains, each assigned to a different processing unit for independent computation.

Technology wise, OpenMP (Open Multi-Processing), a widely used API for shared memory multiprocessing in C, C++, and Fortran, is often employed for parallelisation. Developers annotate their code with OpenMP directives to indicate regions that can be executed concurrently.

The subsequent sections of the paper delve into various aspects of the parallel implementation of Game of Life. In Section 2, a brief explanation of the game's rules and mechanics will be provided. Following this, Section 3 offers a step-by-step breakdown of the algorithm's key components and operations. The subsequent Section 4 will explore strategies for parallelising said algorithm. Section 5 describes the parallel design and implementation of Game of Life. The tests conducted on the implementations are detailed in Section 6. In Appendix I, detailed information regarding the initial state representation (Appendix I.I) and instructions for compiling and executing the source code (Appendix I.II) will be provided.

2. Game of Life

Conway's Game of Life is a cellular automaton that operates on a grid of cells, each of which can be in one of two states: *alive*, represented by a white cell, or *dead*, represented by a black cell. At each time step, the transitions rules displayed in Figure 2, herein explained, are applied:

1. **Birth:** A dead cell with exactly three live neighbours¹ becomes alive (is "born") in the next generation.
2. **Survival:** A live cell with two or three live neighbours remains alive in the next generation.
3. **Death by Isolation:** A live cell with fewer than two live neighbours dies due to underpopulation in the next generation.
4. **Death by Overcrowding:** A live cell with more than three live neighbours dies due to overcrowding in the next generation, as if by lack of resources.

These rules are applied simultaneously to every cell in the grid for each generation. The game progresses in discrete time steps, with each step creating a new configuration of live and dead cells based on the current state of the grid. The resulting patterns can exhibit a wide range of behaviours, including static patterns, oscillations, and complex evolving structures.

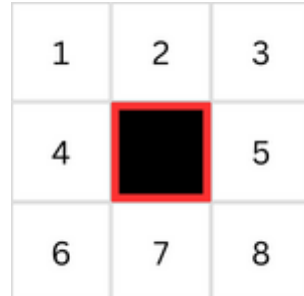


Figure 1: A dead cell (black block in centre with red border) surrounded by its eight live neighbours (white blocks numbered one to eight).

¹A neighbour is defined as any adjacent cell, including diagonals. See Figure 1

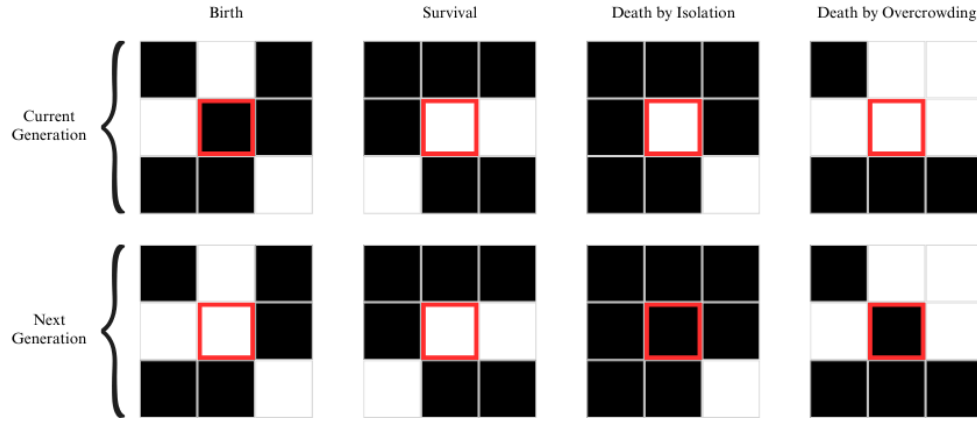


Figure 2: Graphical representation of transition rules. Each column shows the application of **one** rule over **the centre** cell only (marked with a red border), ignoring its effect over the other surrounding cells.

3. Sequential Pseudocode

To provide a clear understanding of the sequential implementation of Conway's Game of Life, the pseudocode detailing the algorithm's key steps is presented in Algorithm 1. It outlines the sequential evolution of the game grid from one generation to the next.

In the algorithm, the `evolve` function takes the current grid state as input and returns the grid state after one generation. It iterates over each cell in the grid, counts the number of live neighbours for each cell using the `countLiveNeighbours` function, and applies the rules of the Game of Life to determine the state of each cell in the next generation. The resulting grid represents the next generation of the game.

This sequential algorithm forms the basis for parallelisation using MPI, where the grid is divided among multiple processes to enable concurrent computation and the game evolution.

```

1: function CONWAY-GAME-OF-LIFE(grid)
2:   ▷ Create a new grid to hold the next generation
3:   nextGrid ← createGridOfSize (grid.size)
4:   ▷ Iterate over each cell in the grid
5:   for cell in grid do
6:     ▷ Count the number of live neighbours for the current cell
7:     liveNeighbours ← countLiveNeighbours (grid, cell)
8:     ▷ Apply the rules of the Game of Life
9:     if cell.isAlive then
10:      if liveNeighbours < 2 || liveNeighbours > 3 then
11:        ▷ Cell dies due to underpopulation or overcrowding
12:        nextGrid[cell.position] ← DEAD
13:      else
14:        ▷ Cell survives to the next generation
15:        nextGrid[cell.position] ← ALIVE
16:      else
17:        if liveNeighbours == 3 then
18:          ▷ Dead cell becomes alive due to reproduction
19:          nextGrid[cell.position] ← ALIVE
20:        else
21:          nextGrid[cell.position] ← DEAD
22:   grid ← nextGrid
23:   return grid

```

Algorithm 1: Game of Life pseudocode.

4. Parallelisation Analysis

4.1. Decomposition Strategy

As discussed in the introduction, this paper proposes to apply domain decomposition. This strategy has been explored in several studies (Ma et al., 2012) with various styles such as “Grid domain decomposition”, “Horizontally striped domain decomposition” and “Vertically striped domain decomposition”.

Domain decomposition is a natural choice because cells are only related to directly adjacent cells. This means that they only need to know about neighbouring locations to compute their next state.

This makes domain decomposition almost trivial. Since it is only required a ghost layer² of adjacent neighbours every time step for each node. All internal cells can be computed with no extra information.

²To be able to compute the short-range interactions, MPI processes need not only access to the data of cells they “own” but also information about cells from neighbouring subdomains, referred to as “ghost” atoms (Lammps, 2023a).

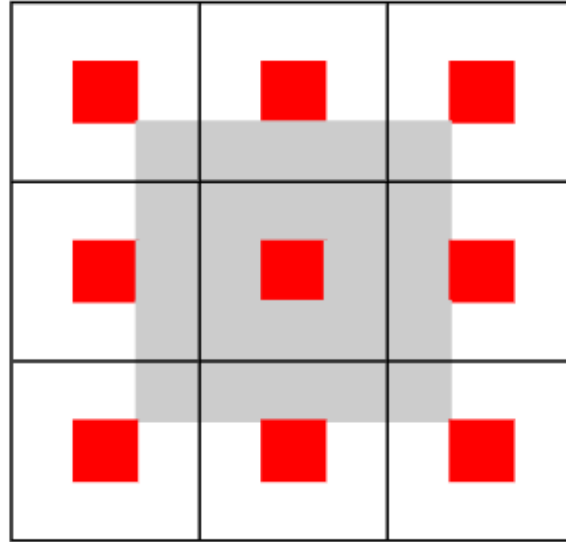


Figure 3: Cells distributed among processes. In light grey are shown the ghost cells and in red, the internal cells.

The implementation of these strategies is not trivial, as it requires that each node (i) is aware of its local grid partition border and (ii) communicates with its neighbouring nodes in order to exchange boundary information. This exchange is crucial for ensuring that each node has the necessary data to compute the next generation correctly.

In this work, grid domain decomposition will be the primary method applied. If time permits Horizontally striped and vertically striped domain decomposition will also be implemented for comparative analysis.

This is due to grid domain decomposition being more promising when it comes to scaling. As the number of cells computed by a process increases, this strategy ensures that the perimeter stays lower compared to the area.

The following subsections provide further explanation on these strategies.

4.1.1. Grid Domain Decomposition

In grid domain decomposition, the grid is divided into smaller sub-grids, and each sub-grid is assigned to a different process. This approach is straightforward and ensures that each process operates on a contiguous portion of the grid. However, if optimisations are applied, such that the computation is focused only where live cells are, it may lead to load imbalance if some regions of the grid contain more live cells than others.

4.1.2. Horizontally Striped Domain Decomposition

Horizontally striped domain decomposition involves dividing the grid into horizontal strips, with each strip assigned to a different process. This approach can help mitigate load imbalance by ensuring that each process handles roughly the same number of rows. However, it may not be suitable for grids where certain patterns or structures span multiple rows.

4.1.3. Vertically Striped Domain Decomposition

Vertically striped domain decomposition divides the grid into vertical strips, assigning each strip to a different process. Similar to horizontally striped decomposition, this method aims to balance the workload across processes by distributing columns evenly. But, it may face challenges with load imbalance if certain columns contain more live cells than others.

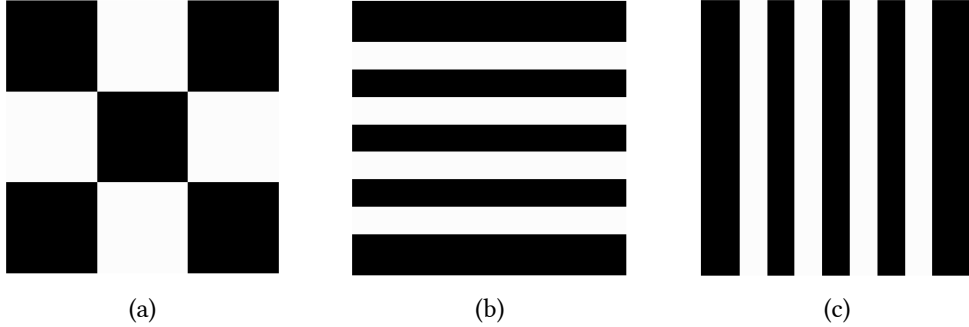


Figure 4: (a) Grid domain decomposition. (b) Horizontally striped domain decomposition. (c) Vertically striped domain decomposition.

4.2. Parallel Algorithm Model

Among all the possible parallel algorithm models, the master-worker model is considered the most applicable to this work. In this model, the master process is responsible for distributing tasks to several worker processes, which perform the actual computations. The master also handles the aggregation of results from the workers.

This approach is well-suited for Game of Life, where the grid can be divided into subdomains, and each worker can process a subdomain independently. The master-worker model ensures efficient task distribution and management, making it an ideal choice for handling the parallelisation of cellular automaton simulations.

While the master-worker model offers an effective framework, it's important to consider the nuances of task assignment within this model. Aside from certain load balancing strategies that could offer advantages if implemented dynamically, dynamic assignment would not provide any further performance improvements due to the use of a large grain size³.

When the tasks are large, the overhead of dynamically assigning tasks can outweigh the benefits, as the processes spend more time managing the task distribution rather than performing computations⁴. Static assignment, where tasks are predetermined and assigned at the start, can be more efficient in such scenarios because it minimises the overhead and maximises the computational throughput. Therefore, while dynamic load balancing strategies can be beneficial in some situations, they are not advantageous for this specific work due to its large grain size.

³In this context, large grain size refers to the significant amount of work assigned to each process (a whole sub-region of the domain).

⁴Since the tasks would likely take a while and not change between processes.

4.3. Communication Model

In this implementation of Conway’s Game of Life, the decision was made to employ the message passing communication model, specifically, utilising the Message Passing Interface (MPI) protocol.

MPI provides a standardised and widely-supported framework for communication in parallel and distributed computing environments. This standardisation ensures portability across different architectures and platforms, facilitating efficient deployment on various parallel computing systems.

Message passing enables efficient communication and data exchange between processes. It allows processes to explicitly send and receive data, promoting seamless coordination and synchronisation between distributed entities.

4.4. Other Optimisation Opportunities

In addition to the strategies discussed in the previous sections, there are other optimisation opportunities available, such as load balancing. Two styles of load balancing explored in this work are: Recursive Coordinate Bisection (RCB) and grid-based methods (Lammps, 2023b).

RCB is a “tiling” method which does not produce a logical regular grid of processors. Rather, it tiles the simulation domain with irregular rectangular sub-boxes of varying size and shape to have equal numbers of particles (or weight) in each sub-box, as in the following rightmost diagram (Figure 5).

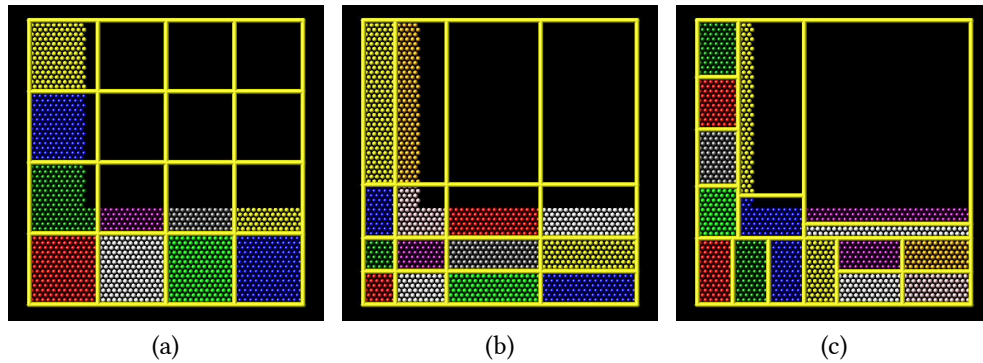


Figure 5: (a) No load balance strategy. (b) Grid method for load balancing. (c) RCB method for load balancing.

Grid balancing, on the other hand, involves organising computational resources in a structured grid-like manner to distribute the workload evenly across the system. Unlike RCB (Recursive Coordinate Bisection), which employs irregular sub-boxes, grid balancing typically involves dividing the simulation domain into a logical grid of processors.

Currently, there are no plans of implementing load balancing. However, if during optimisation it is found that processing can be restricted to areas with live cells (e.g., using a quadtree), then load balancing may be considered.

5. Parallel Design and Implementation

As detailed in the previous sections, the parallel implementation employs grid domain decomposition. Specifically, the chosen algorithm model is the master-worker model. Furthermore, MPI (Message Passing Interface) was selected for communication, using a message-passing approach. At this stage, however, no load balancing strategy has been applied.

The parallel algorithm does not significantly differ from the sequential implementation. Algorithm 2 shows how to apply the sequential implementation across multiple processes to effectively simulate a larger domain.

```
1: function PARALLEL-CONWAY-GAME-OF-LIFE()
2:   ▷ Register MPI Datatypes
3:   ▷ Initialize sub-canvas
4:   ▷ Construct 2d Cartesian Communicator
5:   ▷ Obtain adjacent proceses
6:   ▷ Load initial state
7:   for iter in steps do
8:     ▷ Update ghost cells
9:     ▷ Step all canvas cells
```

Algorithm 2: Parallel Game of Life pseudocode.

5.1. Design

Most of the code from the sequential version was recycled and used here. However, there were significant changes to accommodate the parallel implementation:

- All simulation cells are now two cells wider and two cells taller than required in order to accommodate for a single ghost cell layer on each side.
- A global coordinate system was instated to allow common referencing of the same cell.
- Communication between processes was developed in the following manner:
 - ▷ First, all processes asynchronously send using `MPI_Isend`, utilising a data structure composed of (coordinates.x, coordinates.y, value) to adjacent processes defined by the *Moore Neighbourhood*⁵.
 - ▷ Then, all processes receive the information they were sent.
 - ▷ After that, all processes modify the ghost cells at the border as required.
- A major refactor was conducted to extract common functionality.

⁵In cellular automata, the Moore neighbourhood is defined on a two-dimensional square lattice and is composed of a central cell and the eight cells that surround it.

6. Experiment Design

6.1. Parameters

To evaluate the performance and scalability of this parallel implementation of Conway’s Game of Life, a series of experiments varying several parameters were conducted. The parameters are described in following sections.

6.1.1. Sequential Implementation Parameters

Table 1 describes the parameters applicable to the sequential algorithm.

Property	Description
Grid Size (X)	Width of the grid.
Grid Size (Y)	Height of the grid.
Iterations	Number of generations to simulate.

Table 1: Parameters to the sequential algorithm.

6.1.2. Parallel Implementation Parameters

Table 2 describes the parameters applicable to the parallel algorithm.

Property	Description
Grid Size (X)	Width of the grid.
Grid Size (Y)	Height of the grid.
Process Count	Number of processes used for parallel execution.
Inner Grid Width	Width of the inner grid handled by each process.
Inner Grid Height	Height of the inner grid handled by each process.
Iterations	Number of generations to simulate.

Table 2: Parameters for the parallel algorithm.

Of note it is to state that the grid size is fully determined by the process count and the inner grid sizes via Equation 1 for the case of a square.

$$S_{\text{grid}} = S_{\text{inner}} \cdot \sqrt{n} \quad (1)$$

6.1.3. Common Parameters

Table 3 describes the values used for the parameters in the parallel algorithm. To compare performance, different RLE (Run Length Encoded) patterns⁶ were used. Experiments were conducted on various hardware platforms, listed in the second row of the mentioned table. The number of processes was varied in order to study scalability. Additionally, different sizes for the inner grids managed by each process were tested. All experiments were subjected to a maximum execution time.

⁶RLE (Run Length Encoded) patterns are explained in Appendix II.

The decision behind these choices is explained in Section 6.4.

Property	Values
RLE	("mini.rle", "c4-diag-switch-engines.rle")
Machine	("WGPC", "DGPC", "WGPC+DGPC", "Mica-PC", "toko")
Process count	(4, 16, 36, 64)
Inner Width	(10, 100, 1000)
Inner Height	(10, 100, 1000)
Time Limit (minutes)	3

Table 3: Values for the parameters in the parallel algorithm.

6.2. Metrics

This section details the metrics used to evaluate the performance and scalability of this parallel implementation.

- **Steps executed:** The total number of computational steps completed.
- **Steps per core:** The average number of steps executed by each processing core.

To assess how well the workload is distributed across the computing resources, the following load balancing metrics are used:

- **T Total Execution Time:** The total time taken to complete the computation.
- **T_{cs} Time per cell per step:** The average time spent on each cell for each step. This can be calculated using Equation 2 where $\text{Size}.x$ is the total simulation width, $\text{Size}.y$ is the total simulation height and I is the number of iterations – generations.

$$T_{cs} = \frac{T}{\text{Size}.x \cdot \text{Size}.y \cdot I} \quad (2)$$

- **T_{csc} Time per cell per step per core:** The average time spent on each cell for each step, divided by the number of cores. Computation of this metric can be done using Equation 3, where n is the number of processes.

$$T_{csc} = \frac{T}{n \cdot \text{Size}.x \cdot \text{Size}.y \cdot I} \quad (3)$$

- **C Communication time:** The time spent on inter-process communication. This can be computed as the time spent on step 8 of Algorithm 2. Equation 4 describes how this can be computed and summed for each process as the time from when communication starts up to when communication ends.

$$C = \sum_{\text{Processes}} C_{\text{end}} - C_{\text{start}} \quad (4)$$

- **O Idle time:** The total time that processing cores spend idle. Equation 5 describes idle time where W_{other} represents the time from when the current process finishes communication up to when all processes finish communication.

$$O = C + W_{\text{other}} \quad (5)$$

6.3. Scalability Metrics

The scalability of the parallel implementation, is measured by:

- **Speedup:** The ratio of the execution time of the sequential algorithm to the parallel algorithm.
- **Efficiency:** The speedup divided by the number of processes.

6.4. Objectives

This section explains the reasoning behind choosing the parameters and metrics described in the previous sub-sections.

To determine if the computation time is independent of the initial live cell count, experiments were run with different Run Length Encoded (RLE) patterns: one with a basic input having live cells in the single digits (“mini.rle”) and another in several hundreds (“c4-diag-switch-engines.rle”). Furthermore, the process count was varied while keeping the grid size constant to gauge the weak scaling of the implementation. Conversely, for strong scaling, both the number of processes and the grid size were increased proportionally. Additionally, to assess whether performance trends are highly dependent on, these experiments were conducted on different machines, including desktop PCs, laptops and a cluster (“toko”). Moreover, rectangular inner grid sizes were tested to observe their effects on performance. Finally, communication time was measured to understand its impact on overall performance, and idle time was monitored with the aim of quantifying the periods when processes were not actively computing.

7. References

- Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4), 120–123.
- Lammps. (2023b). *Balance Command*. <https://docs.lammps.org/balance.html>
- Lammps. (2023a). *Communication*. https://docs.lammps.org/Developer_par_comm.html
- Ma, L., Chen, X., & Meng, Z. (2012). A performance Analysis of the Game of Life based on parallel algorithm. *Arxiv*. <https://arxiv.org/pdf/1209.4408>
- Panagiotopoulos, G. (2019). *Conway's Game of Life implemented using MPI & OpenMP*. <https://github.com/giorgospan/Game-Of-Life>
- Papageorgiou, P. (2020). *Game of Life*. <https://github.com/PanPapag/Game-Of-Life>

I. Appendix

I.I. Initial State Representation

The file format utilised to store the initial state is RLE (Run Length Encoding). RLE is a compact and efficient method used to represent patterns or configurations within Game of Life. An example of this format is shown in File 1.

```
1 # Sample RLE file.  
2 x = 9, y = 5, rule = B3/S23  
3 $bo3b3o$b3o2bo$2bo!
```

File 1: Sample RLE file.



Figure 6: The plotted pattern of File 1.

In this format, each line is delimited by '\$'. Within each line, the cells are represented as a sequence of characters where 'b' denotes a black or dead cell, and 'o' denotes a white or alive cell. Additionally, numeric characters are used to indicate the number of consecutive cells with the same state. For example, '3o' indicates three live cells in a row, and '2b' represents two dead cells in a row.

RLE encoding also includes metadata such as the width and height of the pattern, as well as any additional rules that may apply to the pattern's evolution. This metadata ensures that the pattern can be correctly interpreted and simulated within the rules.

I.II. Build and Execution

This project **requires** C++ standard version 23, and at least CMake 3.28. Additionally, it is necessary an MPI-4.1 compatible implementation of MPI.

I.II.I. Build

The steps to compile the previously downloaded source code are outlined below:

1. Navigate to the source root.
2. Create a new directory named `build`.
3. `cd` into that directory.
4. Run `cmake ..`.
5. Run `make -j $(nproc)`.

This last command compiles the project using `mpic++` and `std=c++23`. The resulting compiled binary should be located in `build/src/main/gol`.

The sequential version can be executed with: `build/src/main/gol`.

Meanwhile, the parallel version can be run with:

```
mpirun -n <number of processes> build/src/main/par_gol
```

The **sequential execution arguments** are as follows:

```
src/main.o <filename> <steps_to_simulate> <delay_microseconds>
```

Where the default values are:

- Step count: 0.
- Delay: $50000\mu s = 50ms$.

The final program can be customised modifying the following predefined constants (defines) in the source code:

- `S_WIDTH`: Simulation width.
- `S_HEIGHT`: Simulation height.
- `PRINT`: Verbose logging.