Parallel Implementation of Conway’s Game of Life

Micaela Del Longo, Federico Williamson

Programación Paralela y Distribuida, Licenciatura en Ciencias de la Compuatción, Facultad de Ingeniería, 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. The authors show that decent speedup and efficiency can be achieved using these techniques, however further exploration and optimisations could lead to even lower base compute times.*

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

Contents

1. Introduction

Conway's Game of Life [Gar70] 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 [Pan19; Pap20] 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, its results and their analysis are discussed in Section 7. Finally, in Section 8 a conclusion is provide. In Appendix I, detailed information regarding the initial state representation, instructions for compiling and executing the source code, and additional figures are 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 neighbours[1] 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.

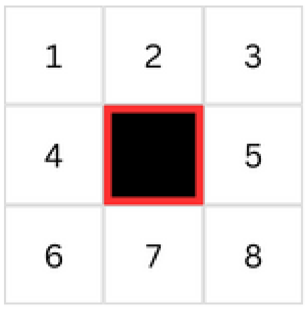


Figure 1: A dead cell (black block in centre with red border) surrounded by its eight live neighbours (white blocks numbered one to eight).

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

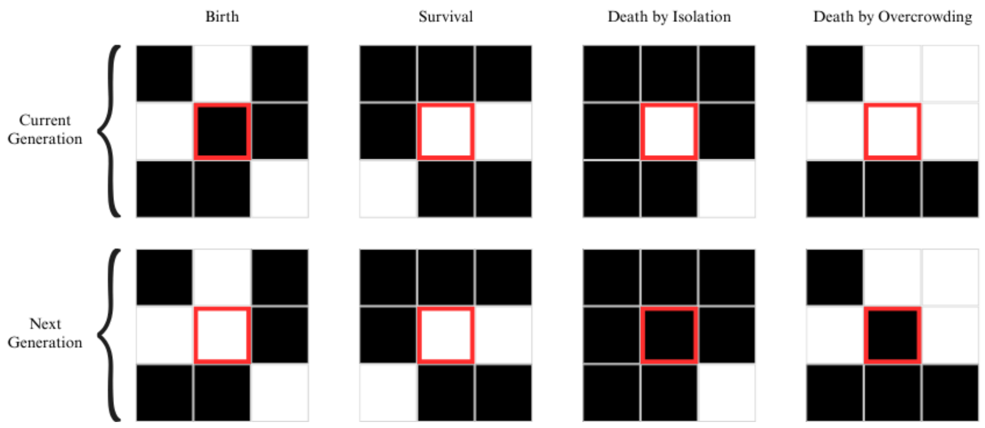


Figure 2: Graphical representation of transition rules. Each column shows the application of **one** rule over **the centre** cell only (marked with a red border), ignoring its effect over the other surrounding cells.

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.

```
function CONWAY-GAME-OF-LIFE(grid)
    ▷ Create a new grid to hold the next generation
    nextGrid ← createGridOfSize (grid.size)
    ▷ Iterate over each cell in the grid
    for cell in grid do
        ▷ Count the number of live neighbours for the current cell
        liveNeighbours ← countLiveNeighbours (grid, cell)
        ▷ Apply the rules of the Game of Life
        if cell.isAlive then
            if liveNeighbours $< 2 \parallel$ liveNeighbours $> 3$ then
                ▷ Cell dies due to underpopulation or overcrowding
                nextGrid[cell.position] ← DEAD
            else
                ▷ Cell survives to the next generation
                nextGrid[cell.position] ← ALIVE
        else
            if liveNeighbours $== 3$ then
                ▷ Dead cell becomes alive due to reproduction
                nextGrid[cell.position] ← ALIVE
            else
                nextGrid[cell.position] ← DEAD
    grid ← nextGrid
    return grid
```

Algorithm 1: Game of Life pseudocode.

4. Parallelisation Analysis

This section delves into the strategies and considerations involved in parallelising Conway's Game of Life. It explores various aspects of decomposition strategies (Section 4.1), algorithm models (Section 4.2), communication models (Section 4.3), and other optimisation opportunities (Section 4.4) aimed at enhancing the efficiency and scalability of the parallel implementation.

4.1. Decomposition Strategy

As discussed in the introduction, this paper proposes to apply domain decomposition. This strategy has been explored in several studies [MCM12] 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 layer[2] of adjacent neighbours every time step for each node. All internal cells can be computed with no extra information.

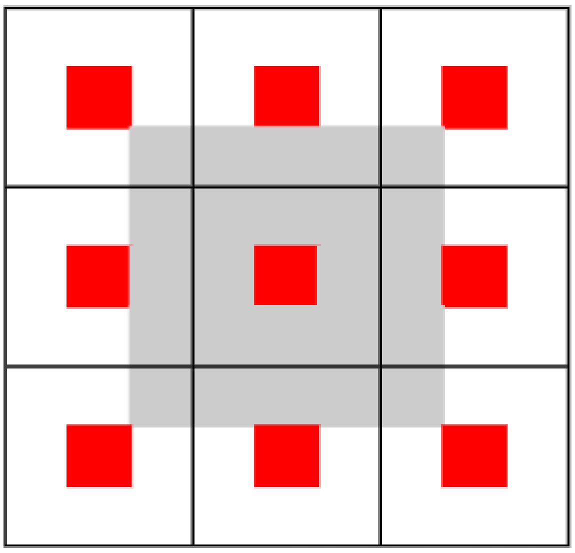


Figure 3: Cells distributed among processes. In light grey are shown the ghost cells and, in red, the internal cells.

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.

[2]To 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"* cells [Koh23].

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.

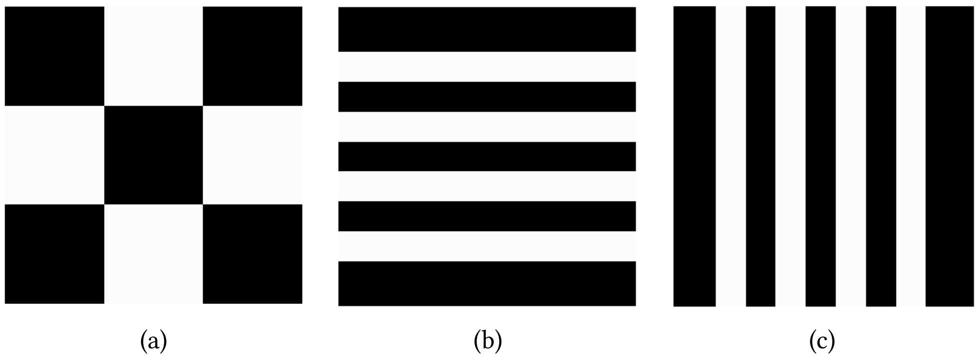


Figure 4: (a) Grid domain decomposition. (b) Horizontally striped domain decomposition. (c) Vertically striped domain decomposition.

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 size[3].

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 per-

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

forming computations[4]. 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.

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 [Ric23].

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).

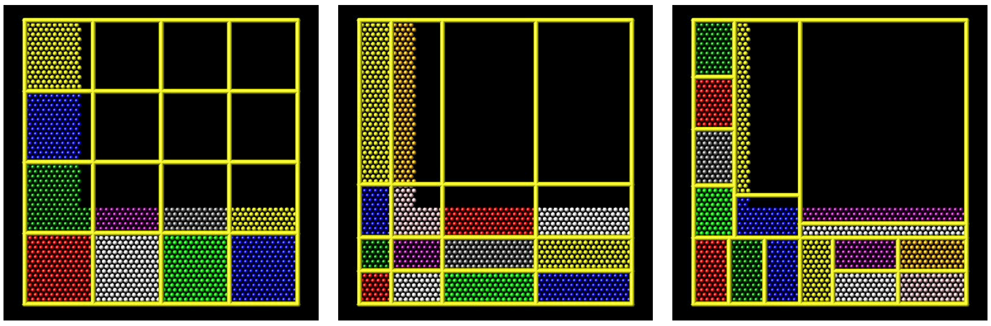

(a) (b) (c)

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,

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

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 quad tree), 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 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.

```
function PARALLEL-CONWAY-GAME-OF-LIFE()
    ▷ Register MPI Datatypes
    ▷ Initialise sub-canvas
    ▷ Construct 2d Cartesian Communicator
    ▷ Obtain adjacent processes
    ▷ Load initial state
    for iter in steps do
        ▷ Update ghost cells
        ▷ 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 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] herein referenced as a coordinate-value pair.
    - Then, all processes receive the information they were sent.
    - After that, all processes modify the ghost cells at their border as required.
- A major refactor was conducted to extract common functionality.

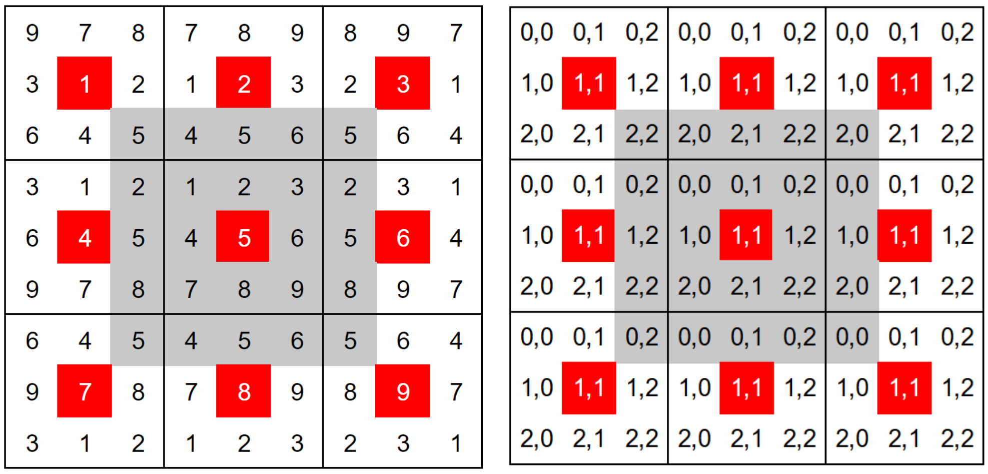
[5]In cellular automata, the Moore neighbourhood is defined on a two-dimensional square lattice and is composed of a central cell and the eight cells that surround it.

5.2. Enhancements

From testing these aforementioned changes, it was apparent that the code worked, but the performance was suboptimal and significantly[6] slower than the sequential version. This was mainly due to a very large proportion of time being taken up by communication and idle time[7].

Taking that into account, the code was changed so that now the processes first build a list of messages to send to each other node and then send an array of coordinate-value pairs. This substantially reduces communication time by reducing the MPI communication overhead.

However, the main optimisation was implementing a cache for *global tag to local coordinate* translation. This cache was used when receiving incoming coordinate–value pairs from Moore-adjacent processes. The former implementation was a brute-force approach that tried all possible *local coordinates* in order to obtain the corresponding *global tag* and return the appropriate mentioned coordinates.



(a) (b)

Figure 6: (a) Shows the “Global Tag” distribution for 'process_count' = 9 and 'inner_width' = 'inner_heigth' = 1. (b) Shows the “Local Coordinates” for 'process_count' = 9 and 'inner_width' = 'inner_heigth' = 1.

Another issue was that the program occasionally hung after execution. This was caused by varying process speeds; during time-bounded execution, some processes would perform an extra iteration and then hit an `MPI_Barrier` inside the main loop. Since other processes had already exited the loop, the program would never finish. The solution was adding more communication: the master process[8] now communicates to the other nodes after each iteration whether another iteration should be performed.

[6]On the order of 62.36%.
[7]The computation time to total time ratio was of about 0.26%.
[8]The one at rank 0 on `MPI_COMM_WORLD`.

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.
Runtime	The total runtime of the simulation.

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 Formula 1 for the case of a square.

$$S_{\text{grid}} = S_{\text{inner}} \cdot \sqrt{n} \tag{1}$$

6.1.3. Run Configurations

Two distinct sweeps were conducted to evaluate the performance of the parallel implementation of the Game of Life under various configurations.

The first sweep focused on comparing different RLE (Run Length Encoded)[9] patterns and the impact of varying process counts on a fixed grid size. Table 3 summarises the properties and values used in Sweep 1.

Property	Values
RLE	• "mini.rle" • "c2-orthogonal.rle" • "c4-diag-switch-engines.rle" • "diag-glider.rle" • "glider.rle"
Processes Count	−1, 1, 4
Inner Width	100
Inner Height	100
Execution Index	0, 1, 2, 3, 4
Time (in minutes)	3

Table 3: Sweep 1 characteristics.

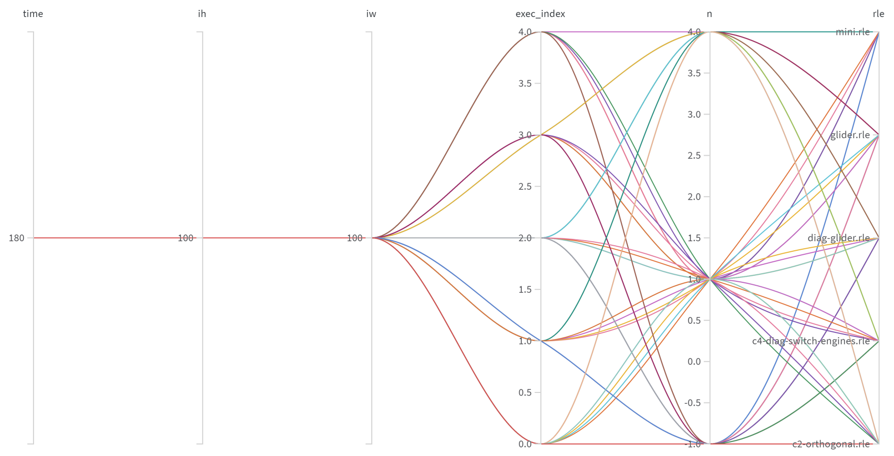


Figure 7: The sweep detailed in Table 3 as a parallel coordinates plot.

[9]RLE files are explained in Appendix I.I.

The second sweep aimed to examine the effects of different grid sizes and process counts on the execution performance using a single RLE pattern. Table 4 below outlines the properties and values used in Sweep 2.

Property	Values
RLE	"mini.rle"
Processes Count	−1, 1, 4, 9, 16
Overall Size Index[10]	1, 2, 3, 6, 9, 20, 30, 59, 84, 187, 270
Execution Index	0, 1, 2, 3, 4
Time (in minutes)	3

Table 4: Sweep 2 characteristics.

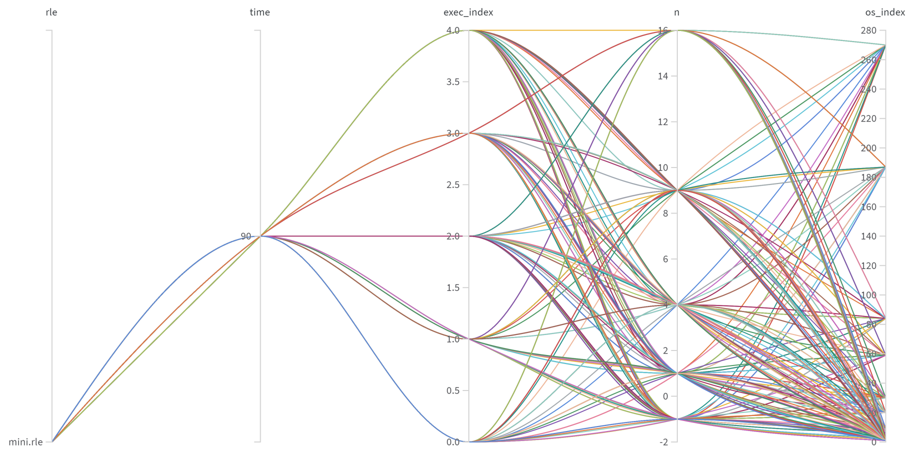


Figure 8: The sweep detailed in Table 4 as a parallel coordinates plot.

In the previous tables (Table 3, Table 4), the '−1' value for *Processes Count* indicates that the sequential version of the algorithm was run. This serves as a baseline to compare the performance of the parallel implementation against the traditional single-process execution. Each test was run five times to ensure the results' reliability and consistency. The Execution Index parameter indicates which run of the test it is, ranging from 0 to 4.

The decisions behind these choices are further explained in Section 6.3.

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.

[10]Overall Size (S) follows the formula $S = 144 \cdot I^2$ where (I) is the Overall Size Index

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

- **Total Execution Time** T: The total time taken to complete the computation.
- **Time per cell per step** T_{cs}: The average time spent on each cell for each step. This can be calculated using Formula 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.

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

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

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

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

$$C = \sum_{\text{Processes}} C_{\text{end}} - C_{\text{start}} \tag{4}$$

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

$$O = C + W_{\text{other}} \tag{5}$$

The scalability of the parallel implementation, is measured by:

- **Speedup** S: The ratio of the execution time of the sequential algorithm to the parallel algorithm. It is calculated with Formula 6, where Ips_s and Ips_p are the number of iterations-per-second of the sequential and parallel model, respectively. Note that we use the parallel over the sequential, as when dealing with iterations *per second,* it becomes necessary to invert the formula. A proof for this statement can be seen in Formula 9, where the *sequential* seconds are divided by the *parallel* seconds.

$$S = \frac{\text{Ips}_p}{\text{Ips}_s} \tag{6}$$

$$S = \frac{I_p}{s_p} \div \frac{I_s}{s_s} \tag{7}$$

$$S = \frac{I_p}{s_p} \cdot \frac{s_s}{I_s} \tag{8}$$

$$S = \frac{I_p \boldsymbol{s_s}}{\boldsymbol{s_p} I_s} \tag{9}$$

- **Efficiency**: The speedup divided by the number of processes.

6.3. Objectives

This subsection explains the reasoning behind choosing the parameters and metrics described in the previous subsections.

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 strong scaling of the implementation. Conversely, for weak scaling, both the number of processes and the grid size were increased proportionally. 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. Performance Analysis

7.1. Execution Environment

The sweeps described previously described in Section 6.1.3 were run on only one laptop ("DGPC"), its characteristics are detailed in Table 5.

Property	Values
Architecture	x86_64
CPU Model Name	11th Gen Intel Core i7-11800H @ 2.30GHz
Thread(s) per core	2
Core(s) per socket	8
CPU(s)	16
L1d cache size	384 KiB (8 instances)
L1i cache size	256 KiB (8 instances)
L2 cache size	10 MiB (8 instances)
L3 cache size	24 MiB (1 instance)

Table 5: DGPC characteristics.

7.2. Analysis

Using the results from the **first sweep** (Table 3), the deviation in iterations-per-second from those obtained with the initial state representation “mini.rle” was calculated. Figure 9 shows that the deviation was found to be less than 10%, leading to the conclusion that computation time is independent of the initial live cell count provided by different Run Length Encoded (RLE) patterns.

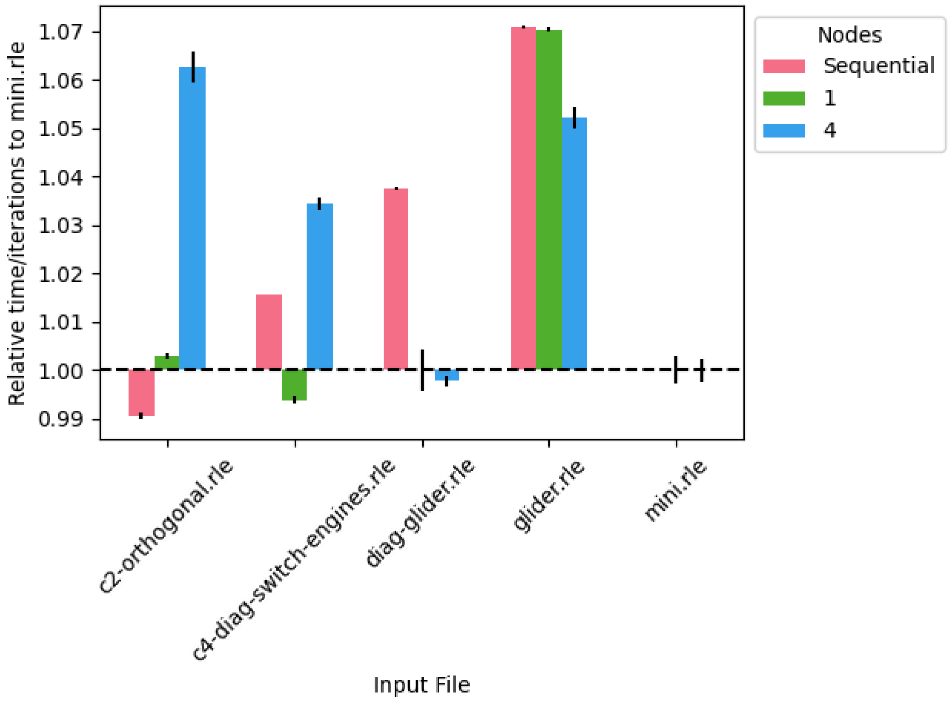
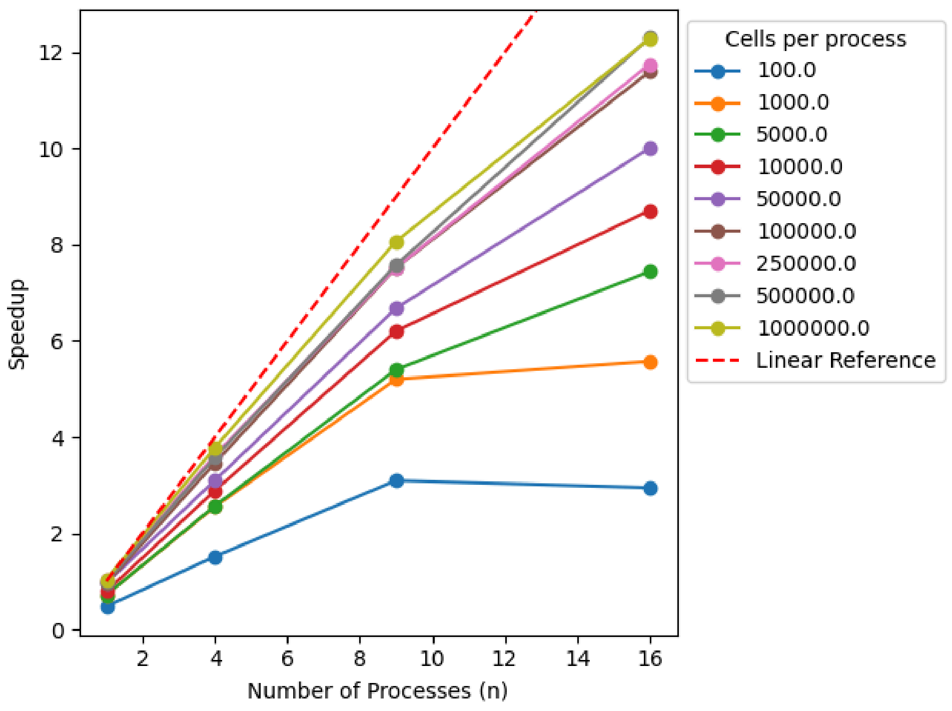


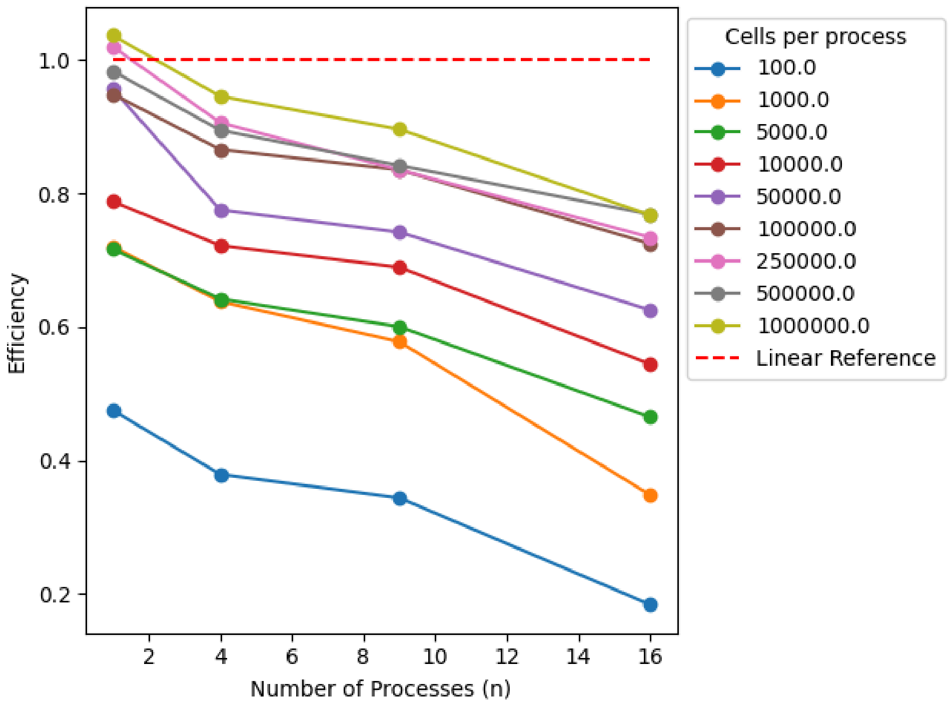
Figure 9: Bar chart comparing the iterations-per-second relative to the “mini.rle”.

The following analysis was done with results obtained from the **second sweep** execution (Table 4).

As described previously, to evaluate the *weak scaling*, the number of processes and the grid size were increased proportionally. In the resulting graphs (Figure 10), it can be appreciated how the speed up improves with the number of cells per process. On the other hand, efficiency decreases with an increasing number of processes. However, the rate of this decrease diminishes as the number of cells per process rises.

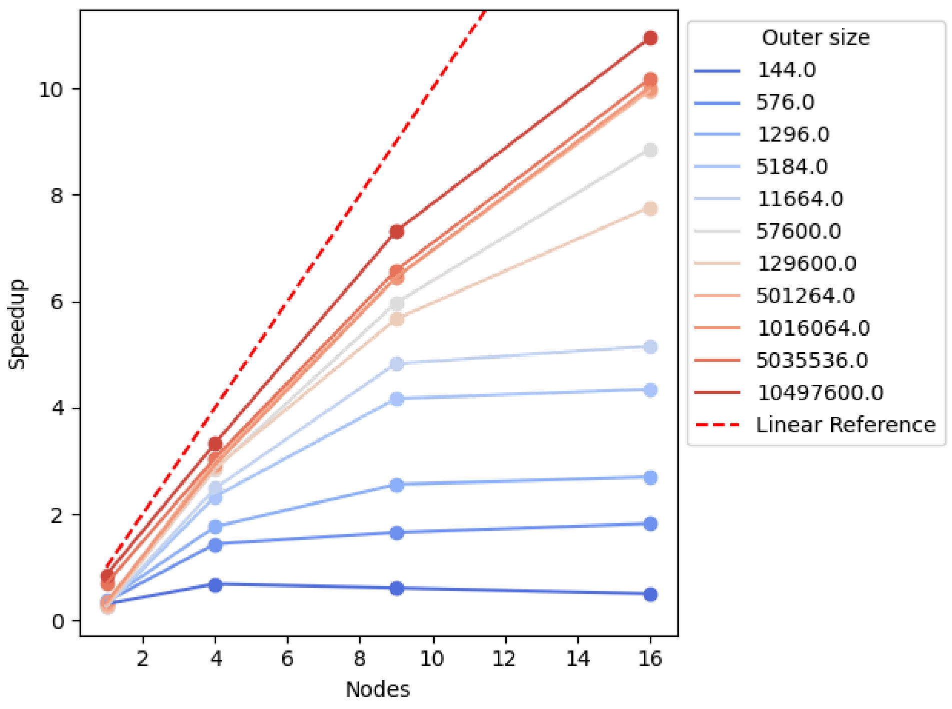


(a)

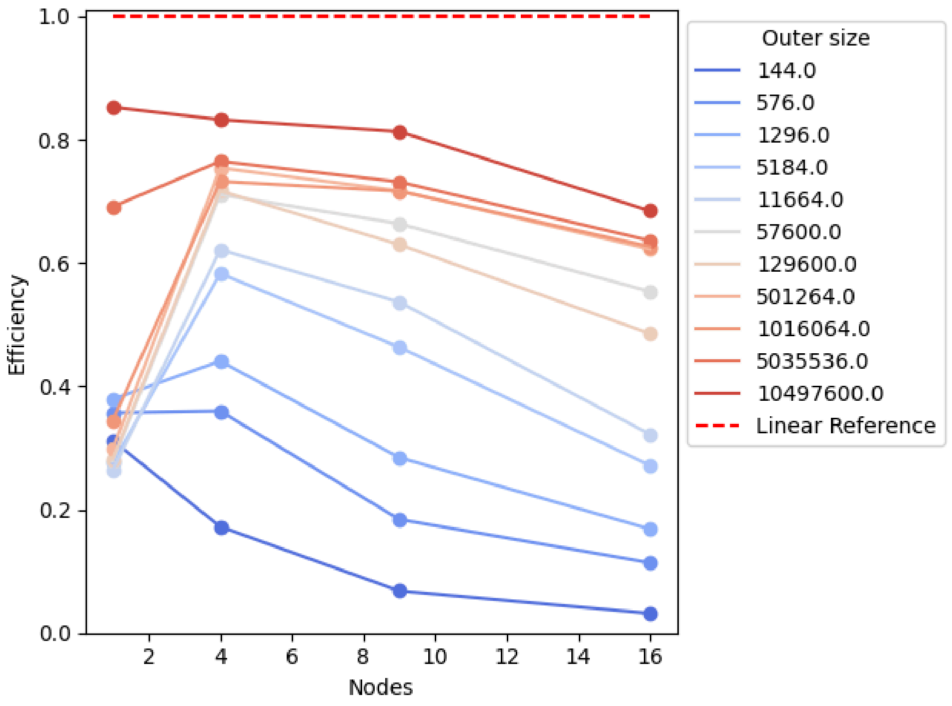


(b)

Figure 10: Speedup (a) and efficiency (b) obtained when evaluating weak scaling.



(a)



(b)

Figure 11: Speedup (a) and efficiency (b) obtained when evaluating strong scaling.

Conversely, to evaluate the *strong scaling* process count was varied while keeping the total grid size constant (See Figure 11). Similarly to weak scaling, the speedup improves with the number of cells per process, and efficiency decreases with an increasing number of processes. With this rate diminishing as the number of cells per process rises.

Figure 12 compares the outer (total) size with the achieved number of millisecond per iterations. The number of milliseconds per iterations increases with the outer size, and it can be seen that increasing number of nodes increases performance (since it reduces the necessary time until the next iteration).

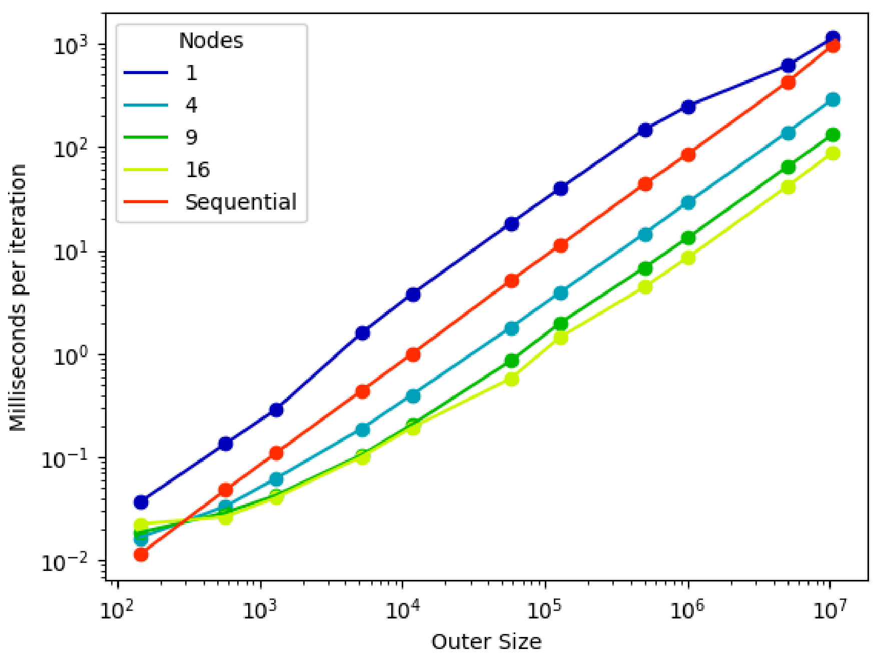


Figure 12: Scatter plot comparing the milliseconds per iterations with the outer size.

Figure 13 illustrates the percentage of time allocated to computation, communication, and idle states across Sweep 2 described in Table 4. It is evident that as the number of processes increases, the time spent on communication and idle states also increases. However, this time does not exceed the time spent on computation.

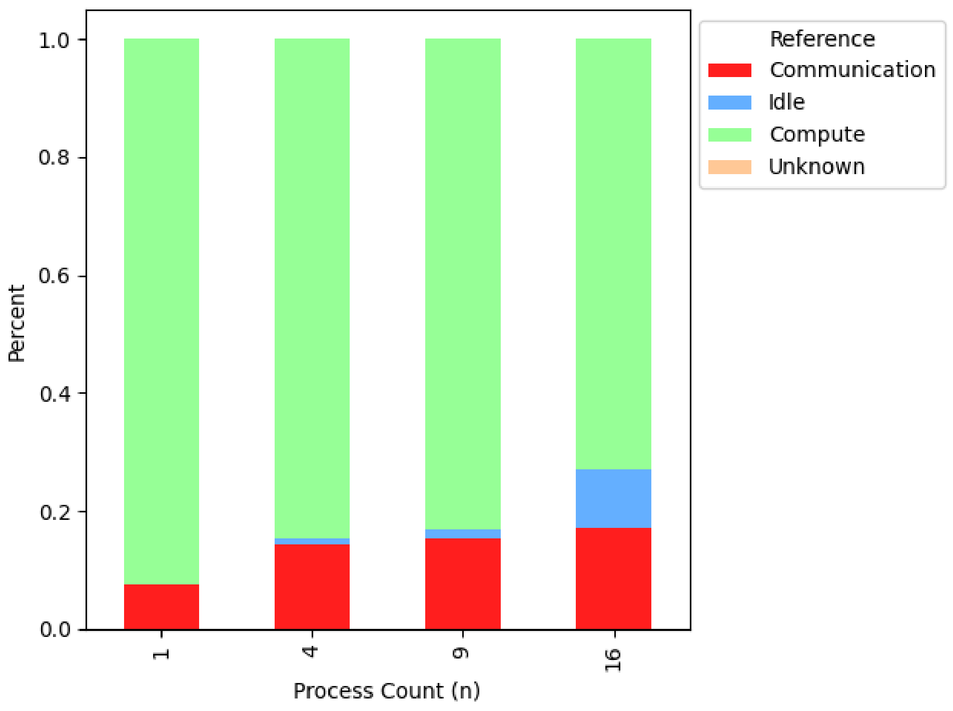


Figure 13: Bar chart comparing allocated to computation, communication, and idle states.

In Figure 13, "Unknown" actually refers to the known time spent storing the data needed for the posterior execution analysis.

8. Conclusions

Overall, the results obtained showcase an improvement over the sequential implementation. However, further experiments are needed to provide a more comprehensive comparison.

We observed fairly good efficiency when the system size was large. This was particularly evident when the inner width and height were large enough that communication was not the predominant time sink. Our analysis of the communication and idle times reveals that as node count increases, communications also grow significantly. Exploring further limits on this aspect would be ideal.

In future implementations, expanding the scope to include experiments across different hardware configurations could offer valuable insights into the system's performance variability.

8.1.1. Future Improvements

To enhance the performance and efficiency of this parallel implementation, several potential improvements can be considered:

Implementing a strategy to skip computations in areas without live cells could significantly reduce unnecessary processing. If this approach is adopted, integrating a load-balancing mechanism would ensure an even distribution of computational workload across processes.

Not computing the state of ghost atoms could streamline the algorithm and reduce computational overhead.

Adding multiple layers of ghost atoms might improve the efficiency of boundary cell computations, potentially leading to better overall performance as more iterations may be accomplished without the need to communicate.

9. References

[Gar70] Gardner, Martin: Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". In: *Scientific American* vol. 223, Springer (1970), Nr. 4, pp. 120–123

[Koh23] Kohlmeyer, Axel: *Lammps - Communication.* URL https://docs.lammps.org/Developer_par_comm.html. — (Accessed: 28/05/2024)

[MCM12] Ma, Longfei ; Chen, Xue ; Meng, Zhouxiang: A performance Analysis of the Game of Life based on parallel algorithm. In: *arxiv* (2012)

[Pan19] Panagiotopoulos, George: *Conway's Game of Life implemented using MPI & OpenMP.* URL https://github.com/giorgospan/Game-Of-Life. — (Accessed: 28/05/2024)

[Pap20] Papageorgiou, Pantelis: *Game of Life.* URL https://github.com/PanPapag/Game-Of-Life. — (Accessed: 28/05/2024)

[Ric23] Richard Berger, Axel Kohlmeyer: *Lammps - Balance Command.* URL https://docs.lammps.org/balance.html. — (Accessed: 28/05/2024)

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.

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

File 1: Sample RLE file.

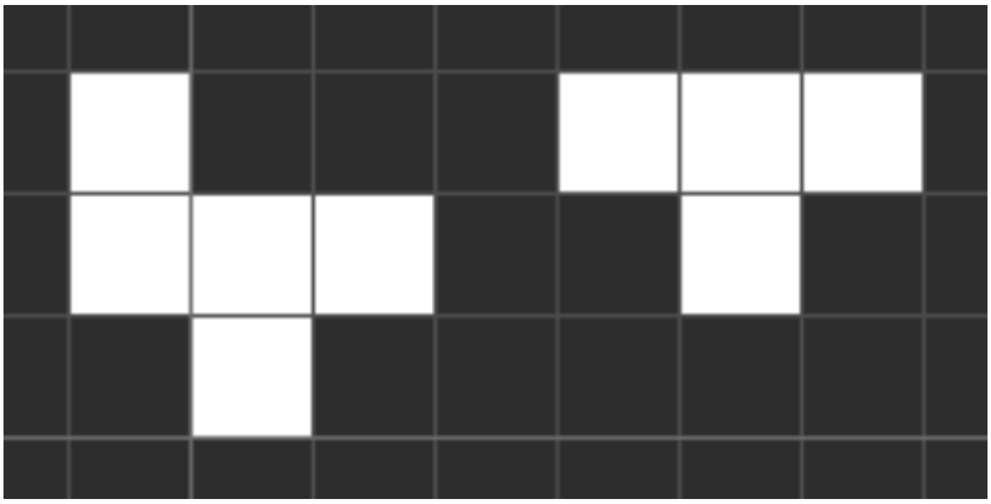


Figure 14: The plotted pattern of File 1.

In this format, each line is delimited by ‘ `$` ’. Within each line, the cells are represented as a sequence of characters where ‘ `b` ’ denotes a white cell, and ‘ `o` ’ denotes a black 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, an MPI-4.1 compatible implementation of MPI is necessary.

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` for the available number of processors. 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.

I.III. Experiments Execution

This section provides instructions on how to set up and run the experiments using *Weights and Biases (WANDB)* for tracking and managing the progress. Thus, having a WANDB account is required, such can be created at “wandb.ai”.

Having previously created a WANDB account, the steps are as follows:

1. Install WANDB locally:

   Use pip to install WANDB by running the following command:

   ```
   pip install wandb
   ```

1. Login to WANDB:

   Run the following command to log in to WANDB and paste in your token when prompted:

   ```
   wandb login
   ```

2. Create a `sweep.yaml` :

   Define your sweep configuration in a file named `sweep.yaml` . This file will contain the parameters and configurations for your experiments.

3. Create the sweep:

   Use the following command to create the sweep in your WANDB project. Replace `<project>` with your project name and `<entity>` with your username or team name:

   ```
   wandb sweep -p <project> -e <entity> sweep.yaml
   ```

4. Prepare the executor environment:

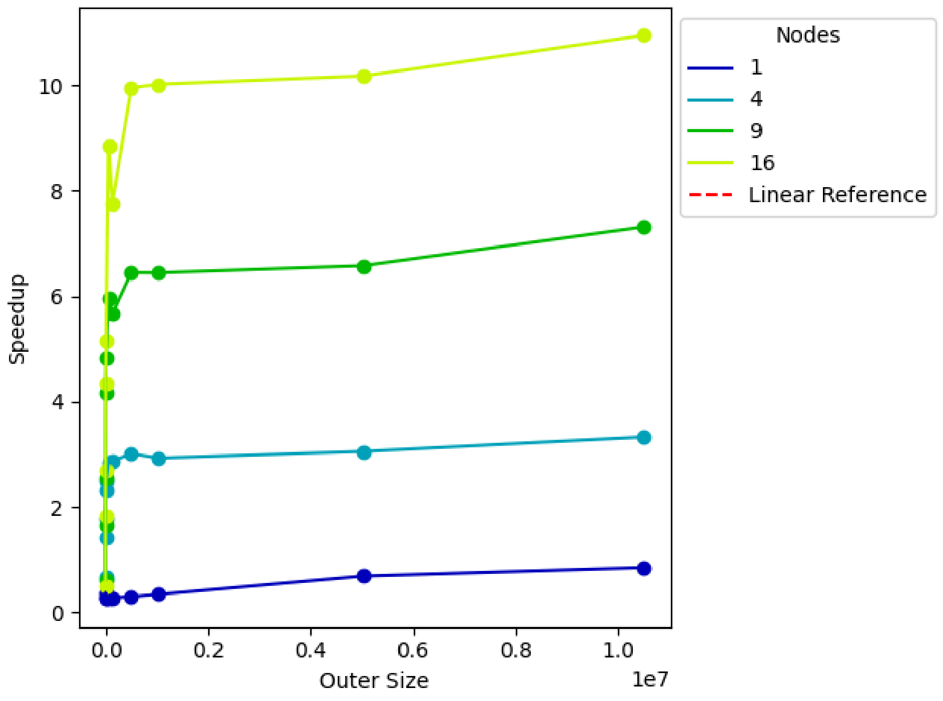
   Create a file named `available_processes.txt` inside the executor directory. This file should list the number of cores available on the machine.

5. Run the sweep agents:

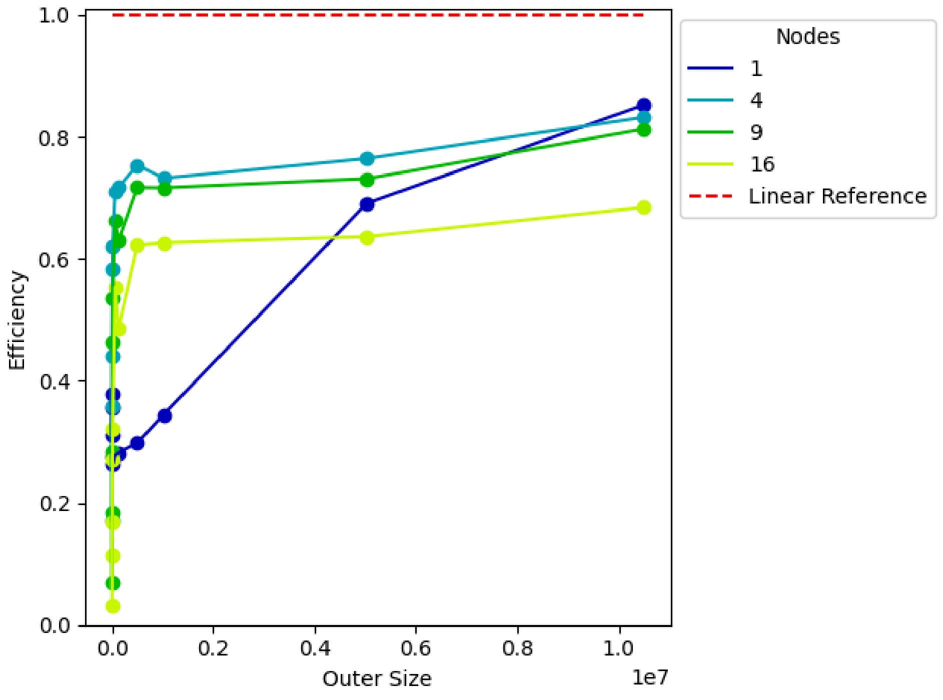
   From the executor directory, run the sweep agent command as many times as needed to match the available cores. Use the `nohup` command to ensure the agents continue running in the background:

   ```
   nohup <sweep agent cmd> &
   ```

I.IV. Additional Images



(a)



(b)

Figure 15: Speedup (a) and efficiency (b) obtained when evaluating strong scaling with respect to system size in sweep 2 (Table 4).