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.

Contents

1. Introduction 2
2. Game of Life 2
3. Sequential Pseudocode 3
4. Parallelisation Analysis 4
   4.1. Decomposition Strategy 4
      4.1.1. Grid Domain Decomposition 4
      4.1.2. Horizontally Striped Domain Decomposition 5
      4.1.3. Vertically Striped Domain Decomposition 5
   4.2. Parallel Algorithm Model 5
   4.3. Communication Model 5
   4.4. Other Optimisation Opportunities 6
5. References 6
6. Appendix 7
   6.1. Initial State Representation 7
   6.2. Build and Execution 7

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, n.d.; Papageorgiou, n.d.) 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 will delve into various aspects of the parallel implementation of Game of Life. In Section 2, an 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 4.1 will discuss various domain decomposition approaches. Within Section 4.2, the Master-Worker model will be proposed as the most applicable parallel algorithm model for the project. In Section 4.3, the decision to use message passing as the communication model will be explained. Lastly, Section 4.4 will address additional optimisation avenues, including load balancing techniques such as Recursive Coordinate Bisection and grid-based methods. In the appendix (Section 6), detailed information will be provided regarding the initial state representation (Section 6.1) and instructions for compiling and executing the source code (Section 6.2).

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 or dead.* At each step in time, the following transitions occur:

- **Birth**: A dead cell with exactly three live neighbours[1] becomes alive (is "born") in the next generation.
- **Survival**: A live cell with two or three live neighbours remains alive in the next generation. This reflects the idea of "survival of the fittest" among neighbouring cells.

- **Death by Isolation**: A live cell with fewer than two live neighbours dies due to underpopulation in the next generation, as if by loneliness.
- **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.

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.

```
function Conway-Game-of-Life(grid)
    ▷ Create a new grid to hold the next generation
    nextGrid $\leftarrow$ 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 $\leftarrow$ FnI[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] $\leftarrow$ DEAD
            else
                ▷ Cell survives to the next generation
                nextGrid[cell.position] $\leftarrow$ ALIVE
        else
            if liveNeighbours $== 3$ then
                ▷ Dead cell becomes alive due to reproduction
                nextGrid[cell.position] $\leftarrow$ ALIVE
            else
                nextGrid[cell.position] $\leftarrow$ DEAD
    grid $\leftarrow$ nextGrid
    return grid
```

Algorithm 1: Game of Life pseudocode.

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

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.

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 (Ma et al., 2012) studies with various styles such as "Grid domain decomposition", "Horizontally striped domain decomposition" and "Vertically striped domain decomposition".

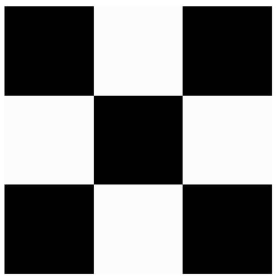
Figure 1: Grid domain decomposition.

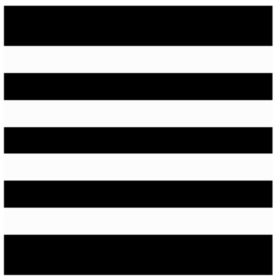
Figure 2: Horizontally striped domain decomposition.

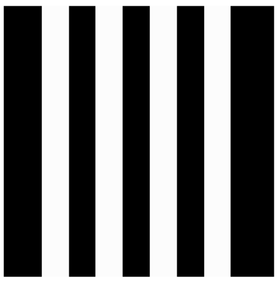
Figure 3: Vertically striped domain decomposition.

The implementation of these strategies is not trivial, as it requires that each node (1) is aware of its local grid partition border and (2) 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.

The following subsections provide an explanation about 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, 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.

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, one processor (the master) is responsible for distributing tasks to several worker processors, which perform the actual computations. The master processor 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. In this context, large grain size refers to the significant amount of work assigned to each processor.

When the tasks are large, the overhead of dynamically assigning tasks can outweigh the benefits, as the processors spend more time managing the task distribution rather than performing computations. 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, utilising the Message Passing Interface (MPI) specifically.

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, fostering 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.

RCB is a "tiling" method which does not produce a logical 3d 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 rightmost diagram above.

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 3D grid of processors.

Currently, there are no plans to implement 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. References

Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, *223*(4), 120–123.

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. https://github.com/giorgospan/Game-Of-Life

Papageorgiou, P. https://github.com/PanPapag/Game-Of-Life

6. Appendix

6.1. 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.

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 o 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.

6.2. Build and Execution

This project **requires** C++ standard version 23, and at least CMake 3.28 or Make 4.3. Additionally, it is necessary an MPI-4.1 compatible implementation of MPI.

For compiling the program, the use of `make` is recommended. The steps to compile the source code with it are outlined below:

1. Navigate to the main directory.
2. Execute `make`.

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

The 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.