MPI implementation of Conway's Game of Life

Micaela Del Longo
Faculty of Engineering
National University of Cuyo (UNCUYO)

Federico Williamson
Faculty of Engineering
National University of Cuyo (UNCUYO)

Abstract

Conway's Game of Life[1], a classic cellular automaton, presents intriguing challenges for parallelization due to its inherently sequential nature. In this work, we propose a parallel implementation of Conway's Game of Life using the Message Passing Interface (MPI) paradigm. Our 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.

I. Introduction

We begin by describing the sequential version of the Game of Life and identifying potential bottlenecks in its parallelization. We then introduce our MPI-based parallelization strategy, which involves decomposing the game grid into smaller subgrids and distributing them among MPI processes. We employ efficient communication schemes to enable inter-process coordination and synchronization while minimizing overhead.

To evaluate the performance of our MPI implementation, we conduct a series of experiments on various parallel computing platforms, ranging from multi-core CPUs to distributed memory clusters. We analyze scalability, load balancing, and overhead characteristics under different problem sizes and MPI configurations. Our results demonstrate significant speedup compared to the sequential version, highlighting the effectiveness of MPI in harnessing parallelism in Conway's Game of Life. **- To be reviewed**

II. Game of Life

Conway's Game of Life, devised by mathematician John Conway in 1970, is a cellular automaton that operates on a grid of cells, each of which can be in one of two states: *alive or dead*. The game evolves according to a set of simple rules based on the concept of generations.

- **Birth**: A dead cell with exactly three live neighbors[1] becomes alive (is "born") in the next generation.
- **Survival**: A live cell with two or three live neighbors remains alive in the next generation. This reflects the idea of "survival of the fittest" among neighboring cells.
- **Death by Isolation**: A live cell with fewer than two live neighbors dies due to underpopulation in the next generation, as if by loneliness.
- **Death by Overcrowding**: A live cell with more than three live neighbors 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 generating 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 behaviors, including static patterns, oscillations, and complex evolving structures. Despite its simple rules, the Game of Life can produce remarkably intricate and unpredictable dynamics, making it a fascinating subject of study in mathematics, computer science, and artificial life.

III. Sequential Pseudocode

To provide a clear understanding of the sequential implementation of Conway's Game of Life, we present pseudocode detailing the algorithm's key steps. The following pseudocode outlines the sequential evolution of the game grid from one generation to the next:

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

```
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 neighbors for the current cell
        liveNeighbors ← FnI[countLiveNeighbors][grid, cell]
        ▷ Apply the rules of the Game of Life
        if cell.isAlive then
            if liveNeighbors $< 2 \parallel$ liveNeighbors $> 3$ 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 liveNeighbors $==3$ 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 Algorithm

In Algorithm 1, 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 neighbors for each cell using the countLiveNeighbors 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 parallelization using Message Passing Interface (MPI), where the grid is divided among multiple processes to enable concurrent computation and evolution of the game.

Bibliography

[1] M. Gardner, “Mathematical games: The fantastic combinations of John Conway's new solitaire game ``life",” *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.