Game of Life: Concurrent and Distributed Implementations

Alex Hetherington & Omer Golan (ss24495 & pr24842)

1 Introduction

This project is focused on implementing concurrent and distributed versions of Conway's Game of Life. In this report, we describe our process implementing, analysing and optimising our various solutions. In the game, the next state of each cell depends on the number of its alive neighbours. Any cell with less than 2 alive neighbours or more than 3 alive neighbours becomes dead, any cell with exactly 3 alive neighbours becomes alive, and any cell with exactly 2 neighbours stays in its previous state. All benchmarks were completed on an Intel Core 5 120U (10 cores, 12 threads), using go's benchmarking feature, using 6 repeats and plotting the mean result using matplotlib.

2 Serial Implementation

Our first implementation iterated over all cells and for each cell we calculated the sum of its neighbouring alive cells by iterating over each of its neighbours and adding 1 if it's alive. For all relevant figure's in this report let $w \in \{0, 255\}^{n \times m}$ with $w_{ij} = 255$ for alive cells and $w_{ij} = 0$ for dead cells. The world is represented as a closed domain, meaning opposite cells are adjacent.

Algorithm 1: Calculate the Number of Alive Neighbours

```
for each cell (x, y) in w do
    n ← 0
    for each neighbour (i, j) of (x, y) do
        if w_ij = 255 then
            n ← n + 1
        end if
    end for
    n_xy ← n
end for
```

Once the neighbour count is calculated, we define the new state of the cell by the function:

$$f(w_{ij}, n) = \begin{cases} 0, & \text{if } n < 2, \\ w_{ij}, & \text{if } n = 2, \\ 255, & \text{if } n = 3, \\ 0, & \text{if } n > 3. \end{cases}$$

Figure 1: Game of Life Initial Rule Set

where n represents the number of alive neighbours. This implementation checks every one of the cell's neighbours at every turn, which is computationally expensive, motivating us to seek a more efficient algorithm.

2.1 Serial Optimisations

To optimise our serial implementation, we used a different approach to calculate the next state of each cell by creating a 2D slice which stored each cell's number of alive neighbours, which we calculated by iterating over each cell and adding 1 to each of its neighbouring cells if it's alive:

Algorithm 2: Calculate Number of Alive Neighbours (Optimised Version)

```
for each cell (x, y) in w do
    if w_xy = 255 then                                  ▷ cell is alive
        for each neighbour (i, j) of (x, y) do
            aliveNeighbours[i][j] ← aliveNeighbours[i][j] + 1
        end for
    end if
end for
```

This way, instead of calculating the number of alive neighbours for each cell, we treat each cell as a neighbour to its neighbouring cells, and add to the sum of neighbour cells only for alive cells. This significantly reduced the time it took for each turn to execute as shown in Figure 2. This reduces the number of operations since instead of looping over each neighbour for every cell, we reduce it to looping over each neighbour for only the alive cells.

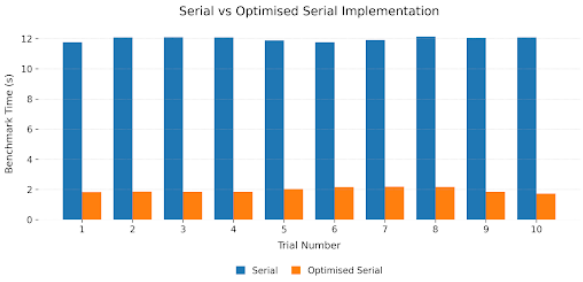

Figure 2: Calculating Next State Function Benchmark

3 Concurrent Version

3.1 First Implementation

Our initial concurrent implementation divided the process of calculating the next world across multiple worker goroutines, which then combined their outputs to produce the next state of the world. We also had two other goroutines: the keypress handler, and the ticker loop. When all the turns have been completed, the main gor-