1 Introduction

Gabriel Siallagan(oi24074) Muhsin Ishaq Hussain(ed24730) In this project, we implemented Conway's Game of Life with a focus on both concurrency and distributed computing. The aim was to first evolve the game state using multiple worker goroutines running concurrently on a single machine, and then extend the system to operate cooperatively across multiple AWS nodes communicating over a network. Both implementations were designed to be efficient, race-condition-free, and deadlock-free, ensuring safe and reliable parallel execution.

In addition, both implementations include several features to enhance functionality: a ticker that reports the number of live cells every two seconds, an SDL display that visualises the game state in real time, and keypress controls that allow the user to pause, quit, or save a snapshot of the current state to a PGM file.

2 Concurrent Section

A goroutine is a lightweight, concurrent function managed by the Go runtime. It runs independently from other goroutines, allowing many tasks to execute simultaneously within the same program.

2.1 Overview of the Parallel Approach

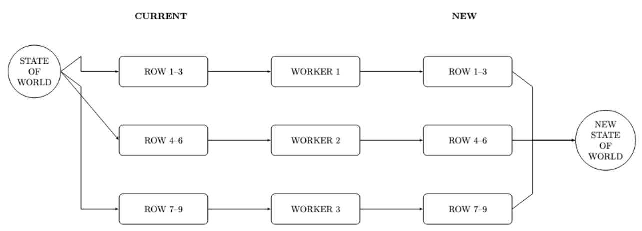


Figure 1: Parallel Worker Design

In the parallel approach to Conway's Game of Life, the world is split into multiple horizontal slices, which are then processed by worker goroutines. The main objective of this design is to distribute workload evenly across a pool of workers, reduce computation, and support scalable parallel execution.

2.2 Goroutines and How They Interact

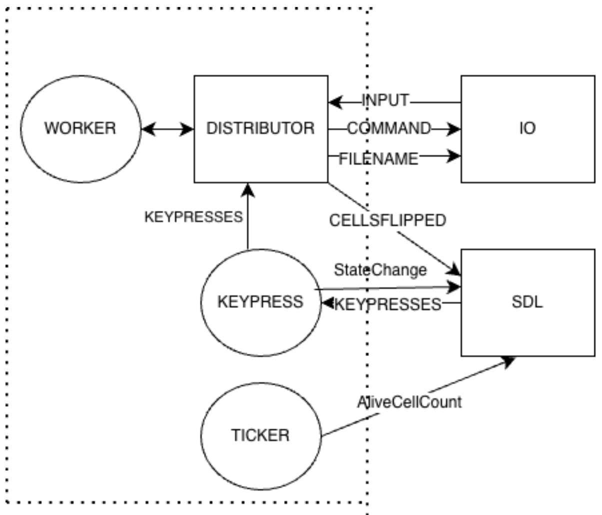


Figure 2: Parallel worker pipeline for Game of Life

The parallel implementation is built around several goroutines cooperating with each other, each responsible for a distinct aspect of the program's behaviour. These goroutines communicate through typed channels to ensure non-blocking progress, consistent snapshots of state, and safe separation of responsibilities.

2.2.1 Worker Goroutine

All workers operate on a shared snapshot of the world while writing to separate output buffers to avoid race conditions. Once all worker goroutines have completed their slice computation, their results are merged into a single world state. In our implementation, workers do not directly write into shared memory; instead, they return their computed slice to the distributor, ensuring deterministic behaviour and avoiding race conditions.

2.2.2 Ticker Goroutine

Every two seconds, the ticker goroutine sends an `AliveCellsCount` event derived from the most recent available snapshot of the world. Because it runs independently in its own goroutine, it introduces no blocking or synchronisation overhead for the main simulation logic.

2.2.3 Keypress Handler Goroutine

This goroutine listens asynchronously for user keypresses such as `p`, `q`, and `s` to pause, quit, or save the game state. It communicates through dedicated channels without mutating shared memory directly. This separation ensures that handling user input is completely decoupled from the worker computation and cannot block simulation progress.

2.2.4 Main Distributor Goroutine

The distributor coordinates the entire simulation. Each turn, it dispatches work to the worker goroutines, collects their results through a channel barrier, and merges them to form the next world state. It also handles pause/quit logic, forwards events to the SDL viewer, and provides consistent snapshots for supporting goroutines such as the ticker and keypress handler. Channels serve as the fundamental communication mechanism, offering safe and lock-free synchronisation between all components.

3 Benchmarking Results and Analysis

3.1 Experimental Setup

All benchmarking of the parallel system was run in a controlled environment on two machines to evaluate performance across distinct hardware platforms: a system running (GoOS: Darwin, GoArch: arm64) with 8 cores and 8 threads (M3 Pro Mac, denoted as "M3 Mac"), and a machine running (GoOS: Linux, GoArch: amd64) with 20 cores (8 P-cores + 12 E-cores) and 28 threads (Intel i7-14700, denoted as "Lab Machine").

A 512×512 board with 1000 turns was used for every test. Multiple worker counts were tested 5 times each, and the average execution time was recorded. This allowed us to evaluate scalability, worker efficiency, and raw performance.

3.2 Results and Analysis

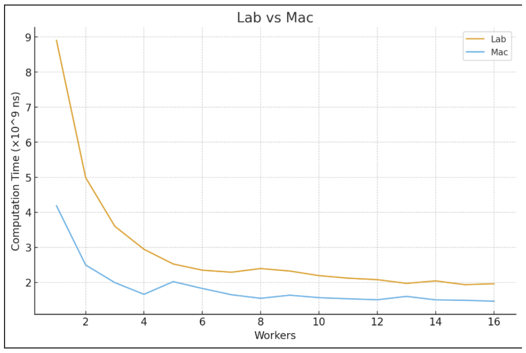


Figure 3: Initial Benchmarking

The graph clearly demonstrates a decreasing trend in execution time as the number of workers increases, with diminishing returns once the number of workers exceeds the CPU's natural hardware parallelism.

3.3 Interpretation and Scalability Analysis

These results indicate that the program benefits significantly from parallelisation. However, communication overhead, memory-bandwidth limitations, and reduced per-worker workload become more significant as worker counts grow large. This behaviour aligns with strong-scaling expectations: speedup improves meaningfully until the number of workers approaches the hardware's true parallel capacity.

3.4 Bottlenecks

Algorithm 1 Using Modulus Operator

```
function COUNTALIVENEIGHBORS($world, x, y, width, height$)
    $sum \leftarrow 0$
    for $dy \leftarrow -1$ to $1$ do
        for $dx \leftarrow -1$ to $1$ do
            if $dx = 0$ and $dy = 0$ then
                continue
            end if
            $ny \leftarrow (y + dy + height) \bmod height$
            $nx \leftarrow (x + dx + width) \bmod width$
            if $world[ny][nx] \neq 0$ then
                $sum \leftarrow sum + 1$
            end if
        end for
    end for
    return $sum$
end function
```

Figure 4: Initial CountAliveNeighbours function

The current naive implementation of `CountAliveNeighbours` introduces a major computational bottleneck. It relies heavily on modulo operations to implement toroidal wrap-around, which are among the slowest arithmetic operations in Go. Because this function executes eight times per cell, for every cell, on every turn, even small inefficiencies accumulate dramatically. Although workers run concurrently, each worker spends most of its time inside this function; therefore, slower neighbour-counting directly reduces scalability.

Memory-bandwidth pressure caused by multiple workers reading the same world snapshot, copying the world each turn, and providing snapshots for auxiliary goroutines also contributes significantly to overhead. As worker count increases, these effects impose an upper limit on speedup.

4 Optimisation

4.1 New `AliveNeighbourCount` Logic

Algorithm 2 Using Modulus Operator

```
function COUNTALIVENEIGHBORSFAST($world, x, y, width, height$)
    $sum \leftarrow 0$
    for $dy \leftarrow -1$ to $1$ do
        for $dx \leftarrow -1$ to $1$ do
            if $dx = 0$ and $dy = 0$ then
                continue
            end if
            $ny \leftarrow (y + dy + height) \bmod height$
            $nx \leftarrow (x + dx + width) \bmod width$
            if $world[ny][nx] \neq 0$ then
                $sum \leftarrow sum + 1$
            end if
        end for
    end for
    return $sum$
end function
```

Figure 5: Optimised CountAliveNeighbours function

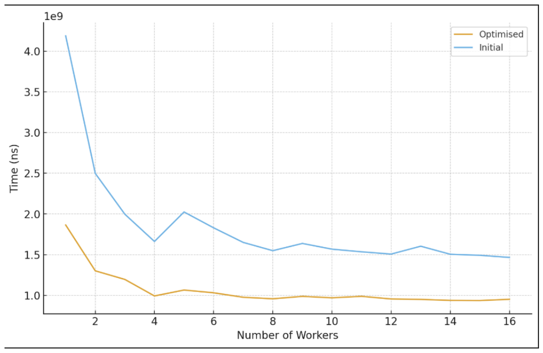
As mentioned in Section 2.3.5, the original neighbour-counting logic relied on modulo operations for edge wrapping. While simple and correct, this approach is computationally expensive when executed across large grids for many turns.

A more efficient strategy replaces modulo operations with explicit boundary checks. Instead of computing

$$(x + i + n) \bmod n,$$

the algorithm uses conditional logic to manually wrap indices when they cross boundaries. Additionally, neighbour coordinate pairs are computed once and reused, reducing repeated index arithmetic. Combined, these optimisations significantly reduce per-cell computation cost and total runtime.

4.2 Benchmarking Results



mac m3 p cores at 4, 2 hardware cores and 2 hyperthreads

Figure 6: Benchmark comparison between optimised CountAliveNeightbours function

The plot shows a clear reduction in computation time across all worker counts, demonstrating that the optimised neighbour-counting strategy improves both parallel performance and scalability.

5 Summary of Parallel Implementation

The parallel implementation of Conway's Game of Life demonstrates strong scalability and efficient work distribution. Worker goroutines independently compute disjoint sections of the world without shared writes, while the distributor safely coordinates merging and communication. The optimised neighbour-counting logic further reduces computation time and enhances scalability. Overall, the system achieves efficient concurrency through careful workload partitioning, channel-based coordination, and targeted algorithmic optimisation.

6 Distributed Section

6.1 System Overview

The distributed system relies on cooperation between a local distributor client, a broker/server, and multiple remote worker nodes. The distributor handles I/O, user interaction, and visual updates, while the server manages global simulation state and orchestrates work across workers via RPC.

As in the parallel implementation, the world is partitioned into horizontal slices. These slices are assigned to worker nodes, which compute updates independently and return results to the server. This separation of responsibilities keeps the distributor lightweight, ensures consistency at the server, and delegates all computation to remote workers.

6.2 Distributed Interactions

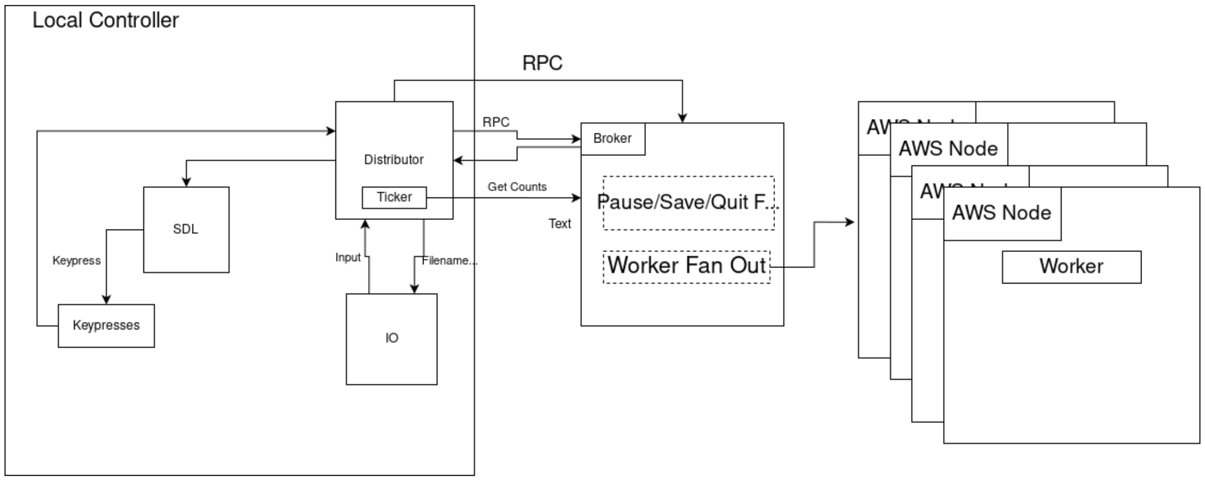


Figure 7: Diagram depicting interaction within system

The distributor loads the initial world, sends it to the server, and listens for GUI-relevant events and user commands (pause, save, quit). The server maintains the authoritative world state, processes user-driven commands, and tracks turns completed so far.

The world is split into slices by the server and dispatched to workers via RPC. Each worker computes the updated cell values for its assigned slice and returns them to the server. Once all worker responses are collected, the server reconstructs the full updated world and sends it back to the distributor for visualisation and event propagation.

6.3 Communication Model (RPC)

RPC provides a synchronous request–response interface that integrates naturally with the Game of Life simulation loop. Workers can be invoked as though they were local function calls while still executing computation remotely. This allows the codebase to remain simple and maintainable while benefiting from distributed parallelism.

6.4 Results and Analysis

6.4.1 Experimental Setup

Benchmarking was conducted on a system running (GoOS: Darwin, GoArch: arm64) with 8 cores and 8 threads. A 512×512 grid with 1000 turns was used for all tests. Worker counts up to 3 were evaluated, each configuration measured 5 times and averaged.

6.4.2 Results

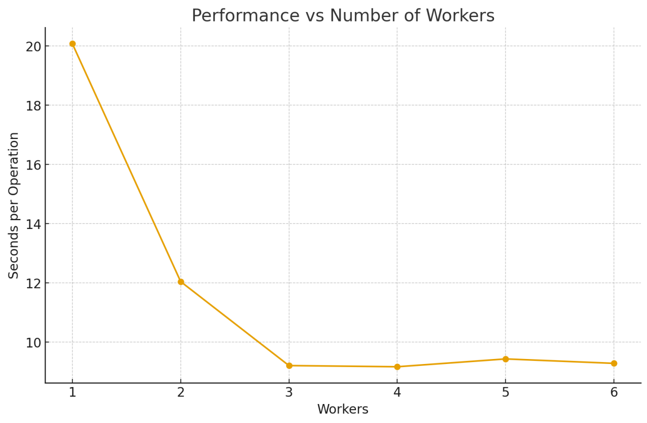


Figure 8: extent of parallelism

The results show a clear decrease in execution time as the number of workers increases, with diminishing returns beyond the machine's natural hardware parallelism. This behaviour mirrors that of the concurrent system and indicates that the program gains significantly from distributed execution.

6.4.3 Bottlenecks

Each worker node processed its assigned world slice using only a single goroutine. This meant that workers could only utilise one CPU core even when more were available, making them the slowest part of the pipeline. As a result, adding more workers offers limited benefit if each individual worker cannot make efficient use of its own hardware.

6.5 Optimisation of the Distributed System

6.5.1 Parallel Distributed System (Extension)

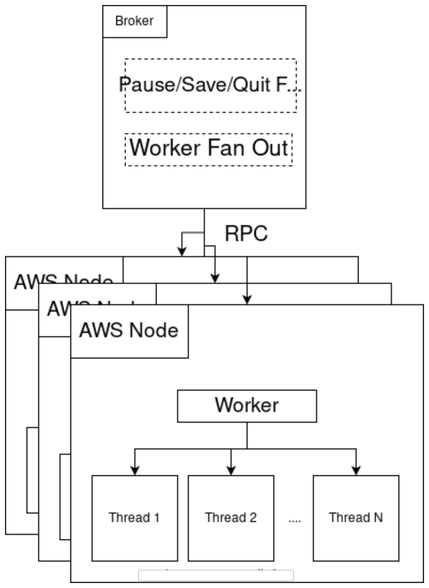


Figure 9: Diagram explaining parallel distributed system

To improve performance, worker servers were upgraded to support configurable multi-threaded execution. Allowing workers to use multiple goroutines internally enabled each to fully utilise its hardware parallelism. This drastically reduced per-turn computation time and improved overall scalability across the distributed system.

6.5.2 Results and Analysis

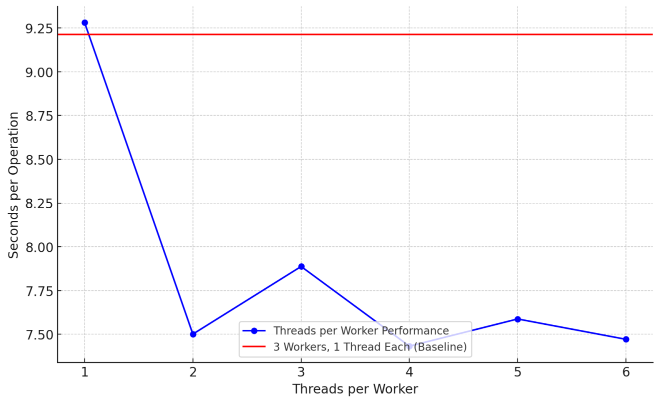


Figure 10: Multithreads Workers v Single-threads at 3 Worker Count

Performance improves significantly after enabling multi-threading within workers. However, gains diminish after two threads, which aligns with the hardware limitations of the t2.medium AWS worker instances used—each providing only 2 vCPUs. Thus, once full hardware utilisation is reached, further threading provides no additional benefit.

6.6 Summary of Distributed Implementation

The distributed implementation offloads computation to multiple remote workers while centralising control logic at the server. Introducing configurable threading within workers further improves performance by allowing each server to exploit available hardware parallelism.

However, the system currently lacks fault tolerance. Because it depends entirely on RPC, failures in workers or network disruptions cannot be recovered from automatically. Adding timeout-based worker recovery, retry mechanisms, or dynamic load balancing would significantly improve the system's robustness and scalability in real distributed environments.