A CSA Simulation on Conway's Game of Life

Kyrian Salas (xv24756) Sergio Fernandez (ng24089) Michael Li (fi24242)

1 Introduction

Conway's Game of Life is a cellular automaton, a zero-player game determined by its initial state on a 2D grid of alive or dead cells. The state of each cell in the next generation is determined by its eight neighbours based on the following rules:

- **Alive Cell:** Survives if it has 2 or 3 live neighbours, otherwise, it dies (due to underpopulation or overpopulation).
- **Dead Cell:** Becomes alive if it has exactly 3 live neighbours.

This report details a brief introduction to a serial implementation, and the design and analysis of two efficient, scalable implementations:

- Parallel Implementation
- Distributed Implementation

1-.1 Serial Implementation

The initial serial implementation involves transforming an input PGM image into a 2D slice of bytes that represents the game world. We then iterate through every cell in the world, evaluating if in the next generation it would be alive or dead based on the rules (1). This is saved to a next world image and then the next generation is calculated from that world. This does not scale efficiently with larger worlds.

2 Parallel Implementation

The parallel model is designed to use multiple threads, dividing the grid and processing multiple cells simultaneously on a single machine. The result is then combined for every turn.

2-A Design and Implementation

2-A.1 Parallelising the Implementation

Our parallel architecture is built on a **distributor-worker** model. A single *distributor* goroutine acts as the central controller, coordinating the work of an $N-sized$ pool of *worker* goroutines. Communication with other parts of the system, such as keypresses and SDL is handled asynchronously via the channel structure *distributorChannels*.

2-A.2 Goroutine Responsibilities

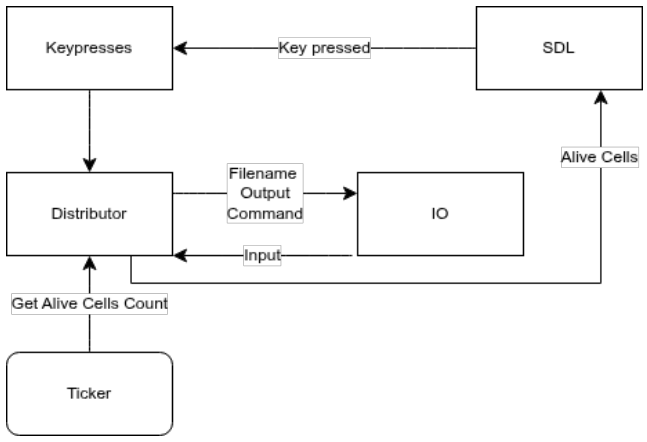

Figure 1: Goroutine interaction

Distributor Goroutine (The "Controller")
The main distributor function handles the central operations of the program.

- Initialises the currentWorld and nextWorld slices, the worker pool, the communication channels such as *taskChan* and the Ticker.
- Loads the initial PGM file into currentWorld (a 2D slice of bytes).
- Handles the main simulation loop, iterating for the specified number of turns.
- Inside the loop, it performs these core actions every turn:
  1. **Work Distribution**: Partitions the grid into N horizontal strips and sends a *task* struct for each strip onto the *taskChan*.
  2. **Synchronisation**: Waits for all N workers to complete by collecting N workers from the *doneChan*.
  3. **Event Handling**: Uses a *select* statement to concurrently handle keyboard inputs (s,p,q) and signals from the Ticker.
  4. **Display Updates on SDL**: Forwards *CellsFlipped* events from the workers to the SDL goroutine via the *c.events* channel.

- After the loop, it saves the final world state to a PGM image and gracefully closes the *c.events* channel.

Worker Goroutines (The processors)
Our implementation utilises a worker pool of N persistent goroutines. These workers operate as consumers, blocking on the buffered *taskChan* to receive a *task* struct. For each task, the worker executes the *calculateNextState* function only on its assigned row partition (from *startRow* to *endRow*). Upon completion, it sends a slice of flipped cells (*[]util.cell*) back to the *distributor* via the *doneChan*.

System Goroutines (IO and SDL)
The *distributor* does not handle file I/O or any of the display rendering itself. Instead, it communicates with two goroutines provided by the framework given, via channels:

1. **The IO Goroutine**: Communicated with through the *c.ioCommand* channel to read/write PGM files.

2. **The SDL Goroutine**: Receives display updates (such as *CellsFlipped* or *TurnCompleted*) from the *c.events* channel and updates it accordingly.

This design allows us to focus on improving and optimising the core solution away from any complicated I/O handling operations, letting the simulation run at its best possible speed.

2-A.3 Concurrency Model and Data Sharing

Concurrency is managed efficiently using a shared-memory model without using mutexes. The *task* structure sent to workers contains pointers to the same *currentWorld* and *nextWorld* arrays. This avoids having to copy the grid across, which is computationally expensive. Our implementation is guaranteed to be race-free due to two design choices:

1. Concurrent Reads: All workers concurrently read from the *currentWorld* slice, which is treated as immutable for that turn.

2. Write Partitioning: Each worker only writes to its uniquely allocated row partition in *nextWorld* slice, which do not overlap.

Since no two workers ever write to the same memory location, there are no data races. The main synchronisation point is the distributor's blocking collection of N results from *doneChan*, which signals the end of a turn.

Implementation Details: Torus World

To handle the "wrap-around" torus world, our initial implementation used the modulus operator (e.g $(i+n)\%n$). We identified this as a major bottleneck in the program's hot-spot, `calculateNextState`, as the modulus compiles to a slow, multi cycle division operation. We replaced this with an explicit border, where edge data is copied before each turn. While this adds a negligible memory overhead (proportional to $O(N+M)$ for an $N \times M$ grid), it replaces the expensive modulus operation with simple, single cycle addition. This optimisation yielded a significant performance increase, as detailed in Section 2-B.

2-B Performance Analysis

This section discusses how we analysed our performance. We will denote the machine running GoOs: Linux, GoArch: amd64, Ryzen 7 PRO 5850U (8 core 16 thread) as Linux Machine. The machine running GoOs: darwin, GoArch: arm64, Apple M1 (8 core 8 thread) as Mac. All tests are run on a 512x512 board size with 1000 turns unless specified otherwise.

2-B.1 Scalability 1

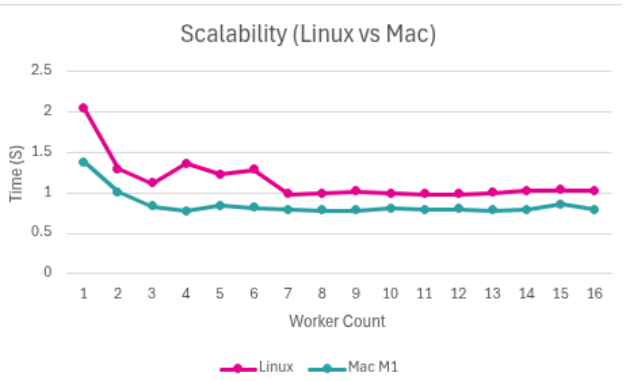

Figure 2: Scaling with Worker Count

Figure 2 compares runtime performance against worker count on two systems. Both systems demonstrate good initial scalability, with runtime dropping significantly as workers are added up to 4 (Mac) and 8 (Linux). After this point, performance plateaus, indicating that the program is now limited by its serial components, as described by Amdahl's Law.

2-B.2 Serial vs Concurrent

As shown in Figure 3, the baseline serial implementation provides a consistent runtime of roughly 3.4 seconds. Our first concurrent implementation, which used the modulus operator, performed significantly worse. We attribute this poor performance to high coordination overhead. Furthermore, this implementation scales negatively (as we add more workers, the runtime increases). This suggests the cost of managing the additional goroutines and their communication is the dominant factor.

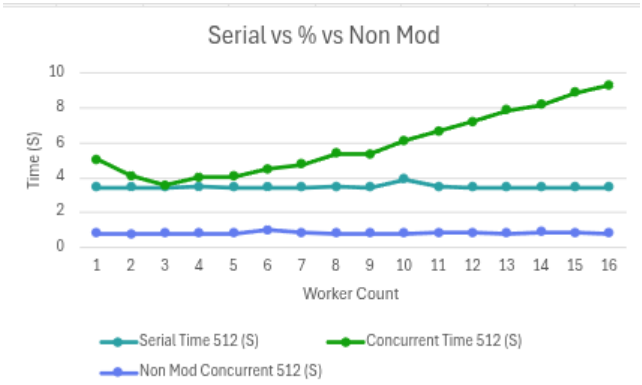

Figure 3: Serial vs Concurrent (Modulus and Non Modulus)

In contrast, our optimised *Non Modulus* implementation is consistently the fastest, finishing in roughly 0.7 seconds. This represents a speedup of over 4.8x compared to the serial version and confirms that our optimisations were highly effective.

2-B.3 Modulus VS Static Border

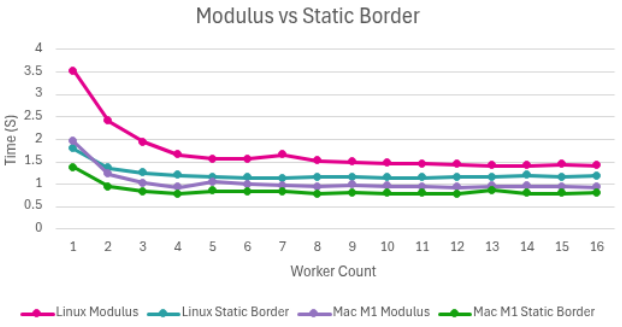

Figure 4: Modulus vs Static

Figure 4 shows a runtime comparison between the *Modulus* and *Static Border* implementations across two systems (Linux and Mac M1). The *Static Border* implementation is consistently faster than the *Modulus* approach on both machines, confirming that the modulus operator was a significant performance bottleneck. In all tests, performance gains are significant from 1 - 4 workers, after which the runtimes plateau. This indicates that the scaling is now limited by serial components, such as synchronisation, which is a clear demonstration of Amdahl's Law.

2-B.4 Scaling with World Size

Figure 5 plots the worker count against deviation from ideal linear scaling (0 on the y-axis). Values below 0 indicate super linear speedup where certain amounts of data per worker fit perfectly into specific cache levels. The big positive spike is explained by the cache line misalignment at that specific worker count.

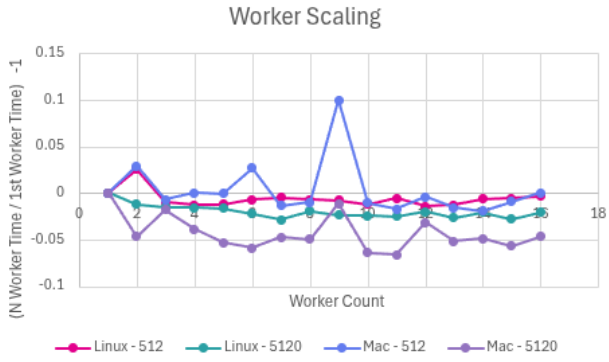

Figure 5: Scaling with world size

2-B.5 Processing time with Scale

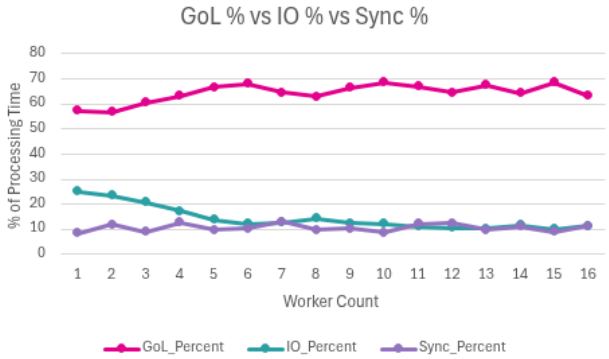

Figure 6: Processing Time per Activity

Figure 6 plots the percentage of time spent on computation (GoL%), I/O wait (IO%), and synchronisation (Sync%). The most critical finding is that Sync% remains flat and low, averaging 8-13%. This proves the design is highly efficient and scalable, as adding more workers does not introduce a synchronisation bottleneck from lock contention or scheduling pressure. As workers are added, the IO% drops from a 25% bottleneck to 12%, as the I/O wait is masked by the parallel computation. Consequently, the proportion of useful work, GoL%, increases from 56% to over 65%.

2-C Optimisation and Future Iterations

2-C.1 Optimisations

1) We initially started by creating and destroying workers for every single turn. We realised that this was not efficient, and decided to make a pool of persistent workers that we could just continue to assign tasks to.

2-C.2 Future Iterations

PProf: We used pprof to identify bottlenecks in our code runtime. The CPU profiles, on average, showed that most of our processing time was spent on *calculateNextState*. This means that further optimisation ef-

forts would be focused on this function.

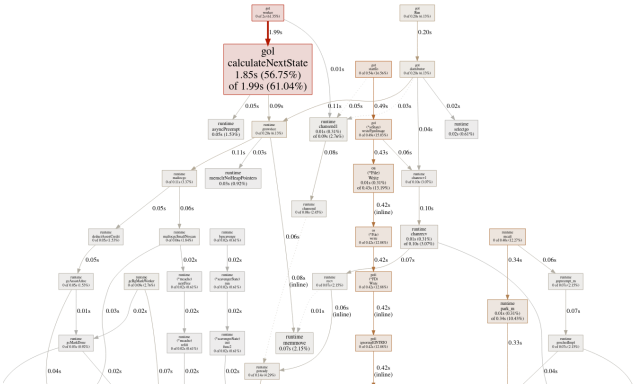

Figure 7: CPU Profile

2-C.3 Alternative Design (Mutex and Condition Variable)

We also benchmarked an alternative design replacing channels with traditional synchronisation. This version used a shared *workerSync* struct, protected by a *sync.Mutex* and coordinated via a *sync.Cond*. The distributor and workers used this struct to manage a shared task slice and a shared results list. Figure 8 shows the

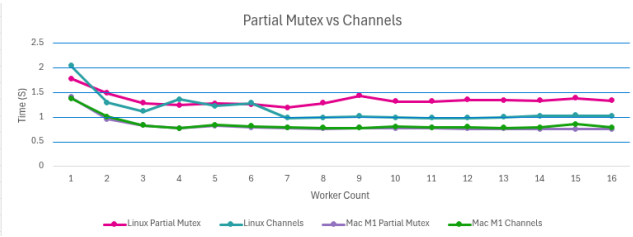

Figure 8: Partial Mutex vs Channels

performance of these two synchronisation models was highly system-dependent. On the Mac M1 (arm64), the *Partial Mutex* implementation was consistently faster than the *Channels* implementation, suggesting lower synchronisation overhead for mutexes on this architecture. Conversely, on the Linux (amd64) system, the *Channels* implementation was consistently faster than the *Partial Mutex* one. This indicates that while Go's channels may offer more portable performance, the raw speed of mutexes can vary significantly by CPU architecture.

3 Distributed Implementation

For the distributed implementation, we use a broker/server and separate nodes. Each node contains a worker with the ability to communicate with the broker, other workers and calculate the next world state. The broker has the ability to communicate with the workers, run the GoL and communicate with the client. The client has the ability to load/save PGM files, and initiate the GoL.

3-A System Design and Implementation

3-A.1 Architecture

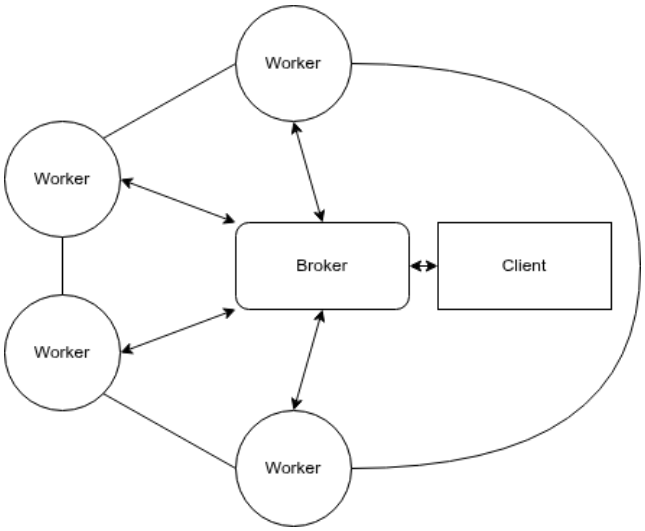

Figure 9: Distributed GoL System (Halo) Architecture.

Our final distributed architecture, shown in Figure 9, is a **fault-tolerant controller-worker** model that uses **peer-to-peer halo exchange** for performance. It consists of three main components communicating via Go's *net/rpc* package.

1. **The Client**: This is the user facing entry point to the program. It connects only to the Broker. It is responsible for loading the initial PGM file, sending the world and turn count to the Broker, and retrieving the final world state at the end of the simulation to store it as a PGM and update the SDL with the final image. It also handles user key presses (p,s,q) by forwarding them to the Broker.

2. **The Broker (Controller)**: The broker acts as the central coordinator and fault monitor. Its primary roles are: 1) Registering new worker nodes as they come online. 2) Distributing the initial world partitions to each worker. 3) Monitoring worker liveness via responses. 4) Orchestrating recovery in case of a worker failure. 5) Signalling the start/stop of the simulation and collecting the final result for the client.

3. **The Workers**: These nodes perform the actual computation. Each worker is responsible for: 1) Calculating the next state for its assigned partition of the grid. 2) Communicating peer-to-peer with its "north" and "south" neighbours to exchange halo rows (its top and bottom rows) after each turn.

3-A.2 Data Communication

Client–Broker

- **On Start**: The client makes one RPC call, sending the initial world (*[][]byte*) and *turns int* to the Broker.
- **During Runtime**: The client forwards key presses to the Broker via RPC.
- **On Finish**: The client retrieves the completed *[][]byte* world from the Broker via RPC.

Broker–Worker

- **On Start**: The Broker calls each worker via RPC, providing its world segment and neighbour addresses.
- **During Runtime**: The Broker monitors worker health and detects failures.

Worker–Worker (Halo Exchange)

- Workers exchange halos every turn: each sends its top row to its north neighbour and its bottom row to its south neighbour.
- Workers block until halos from both neighbours are received before proceeding.
- This peer-to-peer pattern avoids the Broker becoming a bottleneck.
- Each worker also uses internal parallel goroutines to process its segment.

3-B Performance Analysis

3-B.1 AWS vs Local

Figure 10: AWS vs Local Performance

AWS provides scalability as nodes process partitions concurrently, but the local machine offers faster runtime for low work loads due to low latency.

3-B.2 SDL Live View

This is a real time visual mode that displays the cells being flipped, once enabled the client provides a `LiveViewAddr`. Each worker returns a list of `flippedCells`. The broker concatenates these cells and sends them to the `LiveView` server via `LiveViewUpdate`, which turns these updates into `CellsFlipped` events for the SDL. If the SDL were to fail, the broker disables flip collections.

3-B.3 Fault Tolerance

We implemented a fault tolerance mechanism to ensure that the broker still operates even if one of the workers disconnects. Without this, the worker *haloExchange* channels would hang waiting for their disconnected neighbour. There are two possible causes of a disconnection: an RPC call returning an error or connection closed unexpectedly, if this happens then the *failed* flag is raised. The failed worker is removed from the *workerClients*, the world is repartitioned using the previous turn's world, the neighbours are reallocated, the broker sends a *WorkerInitRequest* for each client, and then the turn is restarted. This increases the workload for the remaining workers as their partition of the world is larger and it introduces a minor delay due to the turn being computed again.

3-B.4 System Errors

When encountering errors the program deals with them in the best manner possible, these include:

- **Worker Initialisation failure**: If the worker cannot be initialised it will not be included in the worker pool.
- **Worker disconnection**: If the worker disconnects during execution, then it is removed from the worker pool and the turn is restarted.
- **Live viewer disconnects**: Handled by disabling flip collections.

4 Distributed Optimisation

4-A Parallel Distributed System

We enhanced the distributed worker nodes to calculate their assigned slices concurrently using multiple local threads.

4-A.1 Scalability with Map Size

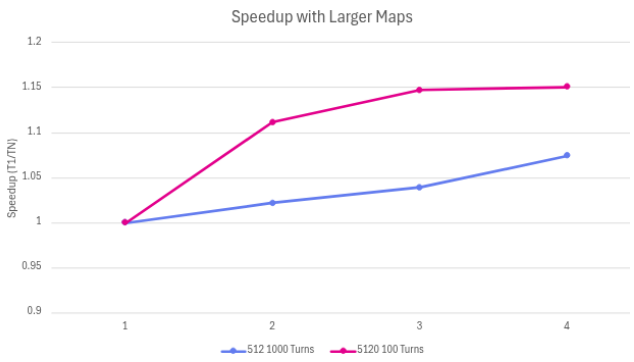

Figure 11: Relative Speed Compared to Map Size

Figure 11 demonstrates that the parallel distributed system achieves a positive speedup (> 1.0) for both map sizes, a significant improvement over the negative scaling observed in the single-threaded distributed version. Notably, the larger 5120x5120 grid exhibits better scalability than the 512x512 grid. This confirms that the system is most efficient when the computational load is high enough to mask the constant network latency overhead.

4-A.2 Parallel Halo vs. Single-Threaded Halo

Figure 12 compares 8-thread vs single-threaded Halo on a 5120 grid. The parallel version is significantly faster at 1 worker due to local concurrency but plateaus immediately. Conversely, the single-threaded version scales effectively, eventually converging with the parallel version at 4 workers. This indicates that network overhead eventually supersedes computation as the bottleneck, neutralising the advantage of local parallelism at higher node counts.

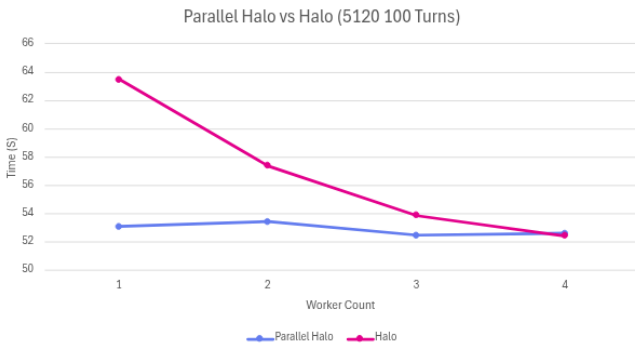

Figure 12: Parallel Halo vs Halo

4-B Halo Exchange

Partitioning the grid introduces boundary dependencies. We address this via `peer-to-peer halo exchange`: workers directly swap top and bottom rows with assigned neighbours, blocking computation until the exchange is complete. This decentralised approach eliminates the Broker bottleneck and offers:

- **Minimal Data Transfer**: Traffic is constant (two rows per worker/turn) and independent of partition height.
- **Scalability**: Direct worker-to-worker communication ensures costs scale linearly with the system size.
- **Fault Tolerance Support**: The Broker can dynamically reassign `topNeighbour` and `bottomNeighbour` targets if a node fails.

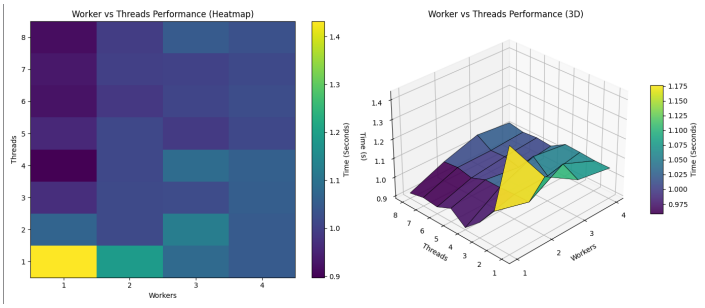

Figure 13: Parallel Halo Runtimes wrt Thread and Worker Count

Figure 13 visualises execution time against Worker and Thread counts. The yellow peak at (1, 1) confirms that serial execution is the primary bottleneck. Increasing threads (y axis on heatmap) provides the most dramatic speedup, rapidly shifting performance into the efficient zone. Conversely, increasing distributed workers (x axis of heatmap) yields diminishing returns, for this specific grid size, the network latency of the Halo exchange outweighs the computational benefit of splitting the task.

5 Conclusion and Future Iterations

5-A Additional Notes

Data used in this report are obtained using Go benchmark tests. Graphs are plotted using Excel and MATLAB.

5-B Further Improvements

- **Traditional memory sharing**: removing the use of channels entirely.
- **Client reconnection**: allowing the client to reconnect and resume ongoing session.
- **Dynamic Worker Load**: adjusting grid partitions based on slice workload i.e if no cells are flipped on that slice then no computation is needed.
- **Backup worker**: performs the task of the disconnected worker.