Parallel and Distributed Game of Life Simulation

Anss Hameed (pg21003) & Louie Sinadjan (bn22907)

Stage 1 - Parallel Implementation

1. Functionality and Design

1.1. Basic Functionality

The Game of Life began with a serial design, updating cells sequentially across a 2D-slice. This process iterates over each cell individually and counts live neighbours, evolving the grid's state one cell at a time using the Game of Life rules.

We later enhanced our implementation by adopting parallel processing. The world is divided among multiple, independent worker goroutines, with each worker responsible for computing the next state of a portion of the grid. The main '*distributor*' function initialises the world and manages the division of work, allowing for parallel processing. Workers then send their results back to the distributor, which then assembles these partial results into the new state of the whole world. We also implemented UI interaction via key presses for pausing ('*p*'), saving images ('*s*') or quitting ('*q*') and a ticker to report the number of alive cells at regular intervals. This parallel approach speeds up the computation by utilising concurrency, a hallmark feature of Go.

1.2. Goroutines

- **Distributor Goroutine:** Acts as the central coordinator, initialising the world and orchestrating the execution of worker goroutines. It also manages user interactions and aggregates results from workers to update the game state.

- **Worker Goroutine:** The implementation uses between one and sixteen worker goroutines. Each worker handles a specific section of the grid, calculating the next of cells in parallel. Computed results are then sent back to the distributor for grid state updates.

- **Ticker Goroutine:** Generates regular time intervals using '*time.Ticker*' to report alive cell counts every two seconds. This operates independently of the game's state, ensuring consistent updates.

- **I/O Goroutine:** Handles file reading for the initial game state setup and file writing for saving the game state. It operates asynchronously to prevent blocking the main game processing.

- **Key Press Listener Goroutine:** Listens for key presses, enabling real-time interaction with the game. It triggers actions for pausing ('*p*'), saving ('*s*') and quitting ('*q*').

2. Critical Analysis

2.1. Acquiring Results – Benchmarking

All tests were repeated 5 times on an 8-core 8-thread Apple M1 (*goos: Darwin, goarch: arm64*) as well as a 6-core 12-thread Intel i7-8700 lab machine (*goos: Linux, goarch: amd64*). All lab machine tests were done on the same machine to maintain integrity and accuracy whilst avoiding potential discrepancies in our results. We used a 512x512 PGM image to benchmark our results, ensuring we push the boundaries on the parallel performance.

2.2. Serial vs Parallel Implementation

We ran benchmark tests across 5 trials on both the serial and parallel implementations.

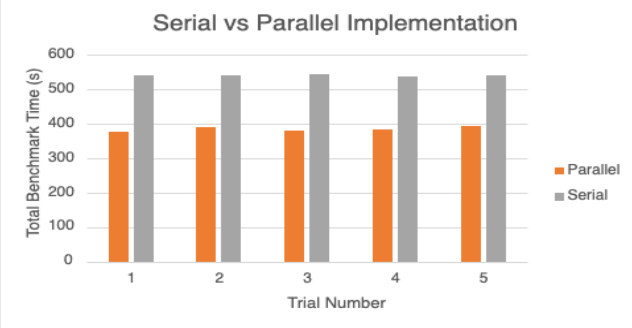

Figure 1: Total benchmark time for serial and parallel implementation across 5 trials

Figure 1 illustrates the comparison between a serial (1 worker thread) and parallel implementation (16 worker threads) across five trials. In each trial, the parallel implementation consistently outperformed

the serial approach, showcasing reduced computation times. This observation aligns with Gustafson's Law (see *Fig.7*), which posits that increasing the number of processors will lead to a linear scale-up in performance for problems with a size that increases with the number of processors. The results depicted by the graph suggest that the parallel simulation effectively capitalises on the additional processing power. This underscores the benefits of parallelisation, where workload can be evenly distributed across multiple threads.

2.3. Algorithmic Efficiency

In this section we focus on performance bottlenecks stemming from algorithms, rather than parallelisation. We illustrate benchmark results obtained from making algorithmic optimisations.

2.3.1. Modulus Operator

The modulus operator's computation expense varies depending on architecture. To test this, we developed two methods for calculating neighbouring cell sums: the original using modulus for edge wrapping and a second approach using conditional statements for edge cases. Both methods were tested on M1 and Intel i7 chips.

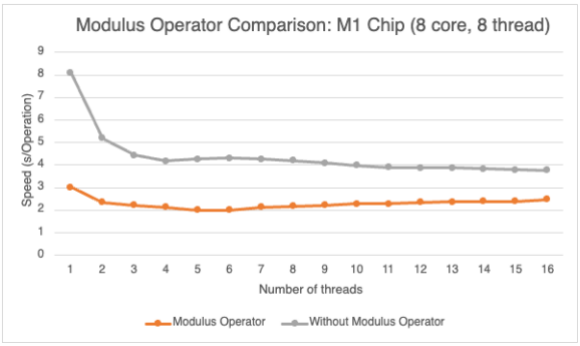

Figure 2: Modulus vs no modulus operator on M1

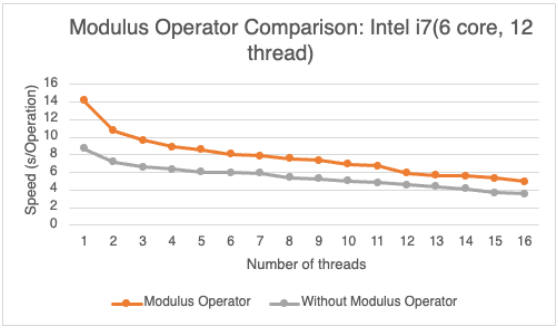

Figure 3: Modulus vs no modulus operator on Intel-i7

The second method is more efficient on an Intel-i7 - substituting expensive modulus operators with simpler arithmetic and conditionals, at the cost of increased code complexity. This exemplifies the trade-off between computational efficiency and code simplicity in software design.

Result: Contrary to our hypothesis of the modulus operator being slower, our tests showed varied results across different architecture. The modulus approach was more effective on the M1 chip while the Intel i7 was the opposite. This highlights the M1's handling of modulus operators, challenging our initial hypothesis about its universal performance impact in multi-threaded environments.

2.3.2. Calculate Alive Cells

Our initial implementation of calculating alive cells (see *Fig.4*) processes the entire grid - amortised time complexity $O(n^2)$. The improved algorithm (see *Fig.5*) only examines the cells which have changed as well as their neighbours, by keeping track of each state - amortised time complexity O(1)

```
Initialize aliveCells as an empty list of Cells

For each cell in the grid (height x width):
    If cell is alive (cell value == 255):
        Add cell to aliveCells

Return aliveCells
```

Figure 4: Original pseudocode - every cell in grid checked – amortised time complexity $O(n^2)$

```
Initialize changedCells as a list of Cells that have changed since the last turn
Initialize aliveCells as an empty list of Cells

For each cell in changedCells:
    If cell is alive:
        Add cell to aliveCells
    For each neighbor of cell:
        If neighbor is alive and not already in aliveCells:
            Add neighbor to aliveCells

Return aliveCells
```

Figure 5: Optimised pseudocode – tracking changed cells - amortised time complexity O(n)

2.4. Hardware Limitations

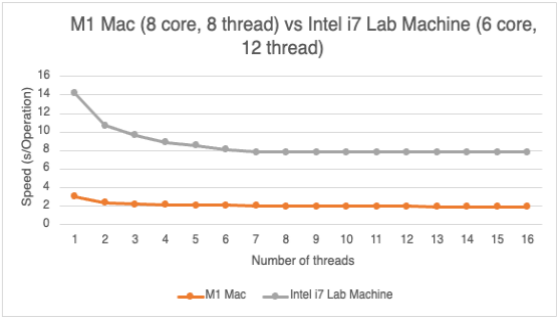

Figure 6: Hardware comparison between M1 Mac and Intel i7 across one to sixteen threads

The trends in *Figure 6* illustrate Amdahl's Law (see *Fig.7*), indicating that parallel computing speedup hits a ceiling due to the program's sequential part. On the M1 chip with 8 threads, performance levels off when all the cores are in use, which is line with Amdahl's concept of diminishing returns at maximum hardware efficiency. Beyond the core limit, no further speedup is observed.

The Intel i7's performance improvements follow Gustafson's Law (see *Fig.7*), showing that parallelism scales with problem size. Hyper-threading (single CPU core processing two threads simultaneously) helps when threads surpass core count but doesn't double core capacity, leading to the overhead seen when threads outnumber cores (see *Fig.6*). These limitations, combined with the CPU's physical and thermal limits, constrain parallel processing's theoretical gains.

Overall, *Figure 6* illustrates that regardless of thread count, the M1's performance has negligible variance in execution. However, the Intel i7 benefits from hyper-threading only up to its physical core limit, after which performance gains taper off.

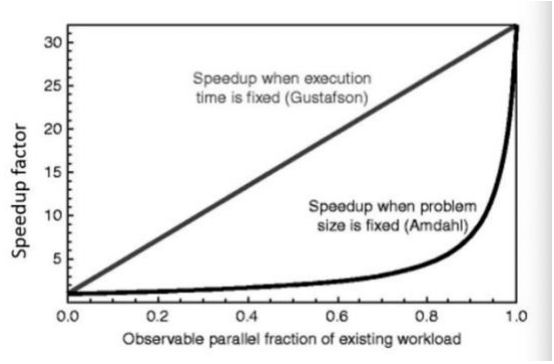

Figure 7: Gustafson, J.L. (2011). Gustafson's Law. In: Padua, D. (eds) Encyclopaedia of Parallel Computing. Springer, Boston, MA.

3. Potential Improvements

3.1. Memory Sharing

Replacing channels with mutexes and semaphores is expected to improve parallelisation outcomes, as seen in the classic '*Producer-Consumer*' problem. This shift leverages mutual exclusion and synchronisation, allowing for direct control over the shared game state. Channels can introduce overhead and potential bottlenecks whereas mutexes and semaphores streamline the coordination among worker goroutines. This results in faster updates to the world and efficiency in handling concurrent tasks. Mutexes and semaphores also align with managing access to shared resources with a reduced likelihood of race conditions. Synchronisation is crucial in an evolving environment like the Game of Life, where the accuracy of each cell's state is paramount.

3.2. Dynamic Load Balancing

Dynamic load balancing ensures that an increasing workload is equitably distributed among worker goroutines, maximizing parallelism, and minimizing idle time. This equitable distribution of work facilitates the efficient scaling of computational tasks, consistent with Gustafson's Law. This posits that enlarging the problem size can lead to better utilization of parallel computing resources and a proportional increase in system throughput.

4. Conclusion

The parallel implementation has shown marked improvements in efficiency and speed, as shown in *Figure 1.* The optimisation to linear amortised time complexity and the performance limitations highlighted by Amdahl's law confirm the benefits and constraints of our approach. Further enhancements in memory sharing are expected to improve efficiency in managing the shared game state.

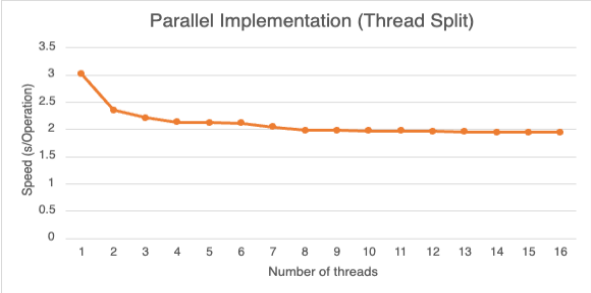

Figure 8 – Speed of s/operation across one to sixteen threads

Stage 2 – Distributed Implementation

1. Functionality and Design

1.1. Basic Functionality

The system is split into three components (see *Fig.1*): the **client-side local controller** – handling user input, **server-side AWS nodes** which act as the workers performing the simulation and a **broker** which acts as an intermediary of communication between the local controller and AWS nodes via RPC (Remote Procedure Calls). Communication is initialised by the local controller connecting to the broker via RPC to start the simulation which in turn returns the final world once the broker has finished. Then, the broker connects to multiple workers to give them slices of the world, which are then appended to each other after evolution. The dimensions of each slice are dependent upon the total number of active workers, such that the workload is proportionately allocated. The files '*stubs.go*' and '*worker.stubs.go*' interlink communication between components via RPC with a '*Request-Response*' design to exchange data.

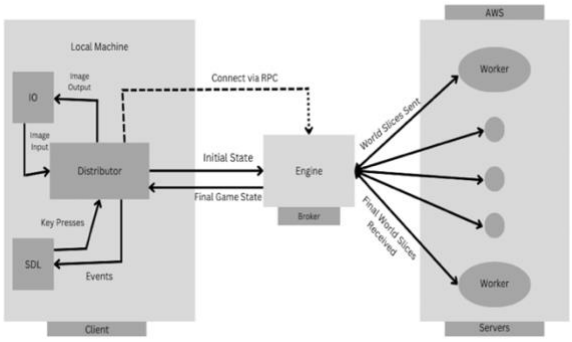

Figure 1 – Distributed System Diagram

1.2. System Components & Design Justification

The distributor (client) is designed solely for task management, excluding Game of Life logic. It serves as the UI, handling I/O and interacting with the SDL subsystem for rendering. The distributor utilises RPC to issue commands to the broker. This approach **abstracts** complexities of network communication.

Clear role separation is maintained by assigning distinct roles to each component. The broker's central position as the engine, orchestrates the workflow of the system via simplification of task distribution and result collection processes. It manages the incoming RPC from the distributor and delegates tasks to the available workers on the AWS nodes, then results are collated in order and returned to the distributor. The broker **centralises** system management, coordinating all computation.

The workers (servers) located on AWS nodes, are responsible for executing the Game of Life logic received from the broker. The use of multiple workers enables the system to distribute the computational load, which can be scaled up or down by adjusting the number of AWS nodes in use. The **scalability** ensures the system can handle varying workloads without compromising performance. **Robustness** is maintained with defined RPC stubs allowing for redundancy and failover; if one worker fails to connect to the broker, others can continue processing.

2. Data Transmission

Referring to the architecture of the broker and server systems in *Figure 1,* when client-initiated termination commands are received, a signal is transmitted to a global boolean channel designated for shutdown operations. The broker and workers each have a goroutine within their main functions, which monitor this channel. Upon receiving a termination signal, the goroutine calls the '*os.exit*' function to cease operations.

Upon pressing '*q*', the client closes without disrupting the server which involves communicating a shutdown signal to the broker. The key presses involve data transmission for command execution and state reporting between the three main components. Moreover, global state information is periodically exchanged, maintaining synchronisation across the system. The RPC mechanism abstracts network communication, allowing the client to invoke methods on the broker as though they were local calls.

Furthermore, mutex locks help synchronisation amongst the distributed components, ensuring variables being accessed during key presses are not simultaneously being updated by 'EvolveWorld'. This allows key presses to be in a goroutine and accept inputs at any point during the simulation.

3. Potential Scaling with Additional Distributed Components

Enhancement of the broker's capabilities is critical for managing a higher volume of RPC and tasking assignments efficiently. Integrating advanced routing algorithms and robust error handling mechanisms, such as retry logic for failed RPC or task reassignment from unavailable nodes, will fortify the system against potential failures.

Integrating cloud scalability features like AWS auto-scaling can dynamically adjust the number of active worker nodes based on the current workload. This feature can be streamlined in the logic for distributor for optimal resource use.

Lastly, deploying AWS nodes across different geographical regions can significantly reduce latency. This would require adding location-aware routing logic in the broker, enabling the system to direct tasks to the nearest available node.

4. Effects of Component Disappearance

Disruptions in worker processes can halt the program's execution, preventing the iteration through game turns and inhibiting the live view of the simulation. While the remaining workers can persist in evolving the world state, the cessation of the broker's functionality necessitates a restart of both the broker and the local controller. Error detection for fault tolerance could ensure continuous simulation by redistributing tasks among active workers upon interruptions.

In the event of the broker's unexpected termination, the entire program ceases operation. Nevertheless, the worker processes maintain their listening state and are equipped to continue processing once the broker and local controller are reactivated.

5. Extensions

5.1. SDL Live View

$$FlippedCells = CurrentWorld \oplus NextWorld$$

Figure 2 – Equation for finding flipped cells using XOR

The SDL Live View is updated dynamically through the dispatch of '*CellFlipped*' and '*TurnComplete*' events to a dedicated channel. These updates are orchestrated by a millisecond-frequency ticker, '*tickSDL'*, that prompts the '*GetCellFlipped*' function within the broker. This function identifies state-changed cells, returning an array of structs with cell coordinates and turn numbers, due to RPC's inability to pass channels. The local controller processes this array, emitting a '*CellFlipped*' event for each change and a '*TurnComplete*' event after the array has been fully processed. This mechanism allows for continuous SDL updates, triggered by the '*tickSDL*' and managed via RPC.

To discern which cells have changed from the last state to the next, the broker performs a logical XOR operation between the corresponding bytes of the two world states, '*g.LastWorld'* and '*g.World' (*see *Fig.2)*. XOR identifies changed cells between the two states, where non-zero values indicate a flip. Due to the world being a 2D byte slice, XOR cannot be applied directly to the slices. Instead, the broker iterates through each cell, applying the XOR operation on a cell-by-cell basis to detect changes.

5.2. Fault Tolerance

Upon the pressing of the '*q*' key, the global boolean variable '*Continue*' is set to true within the quit server function and will retain this state unless the broker restarts. During the initiation of the '*EvolveWorld*' process, the turn counter is set to zero and the global world state is synchronised with the requested world state, but only if '*Continue*' is false, indicating the server's initial run. Consequently, the processing of turns commences from the point at which the server was previously terminated.

Before populating SDL with cell states, the '*Continue*' variable is verified via RPC to the '*GetContinue*' function, which retrieves the status of '*Continue*' along with the global world state. If 'Continue' is affirmed to be true, '*CellFlipped*' events are sent based on the last seen world, and the local turn counter within the broker is aligned with the global turn counter to ensure consistency across sessions.

6. Critical Analysis

All tests were completed on an 8-core 8-thread Apple M1 and a 6-core 12-thread Intel i7-8700 lab machine. Benchmarking on our distributed system consisted of multiple *'m5.large'* AWS instances. Four AWS instances were each assigned a worker, and one instance designated as the broker. We benchmarked on one lab machine to avoid discrepancies on AWS over one session to avoid results being skewed from varying network traffic. All tests completed 1000 turns on a 512x512 PGM image.

6.1. Local Distributed Testing

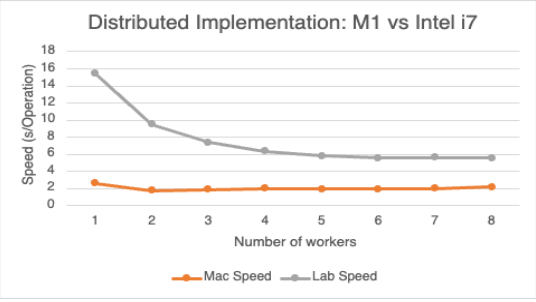

Figure 3 – Local Distributed Comparison with one to eight workers: M1 Mac vs Intel i7 Lab Machine

As the number of workers increase, the speeds of operations on the Intel i7 decline significantly (see *Fig.3*), proving the bottlenecks when optimising less workers. On M1, the overhead of implementing another worker is similar to the simulation speed reduction. Hence, the execution time does not decrease with added workers. Whereas the i7's reduction in execution time can be attributed to the fact that the Game of Life operations were originally slow, making each worker have more utility.

6.2. AWS vs Local

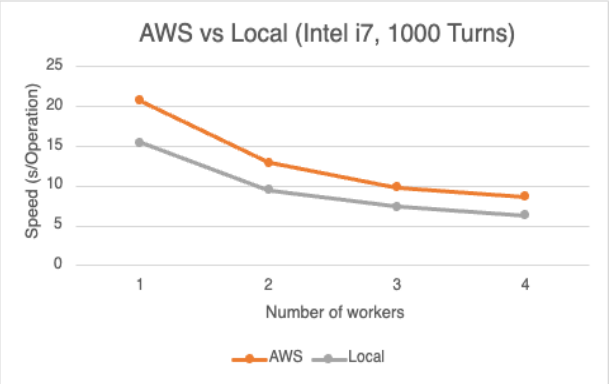

Figure 4 – Speed comparison over one to four workers between AWS instances and local machine

The trend of speed decreasing as number of workers increases (see *Fig.4*) is expected, as each world slice is being distributed across more workers to compute the logic concurrently. The slower AWS performance is due to the network overhead from AWS server connections. Geographical distance causes latency and bottlenecks, slowing AWS operations more than local execution. This data indicates that server proximity and network latency affect the performance of network-heavy distribution.

6.3. SDL vs No SDL

Figure 5 – Speed comparison between SDL and no SDL on the Intel i7 lab machine

The simulation runs slower with SDL enabled (see *Fig.5*), which can be attributed to the overhead introduced by the additional graphical processing. Implementing SDL locks the current state, retrieves '*CellFlipped*' events, iterates over cell updates and then updates the display accordingly. Mutex locking ensures that the update process does not conflict with other operations but contributes to the decreased speed when SDL is active. The combined latency of these steps results in a significant slowdown.

7. Potential Improvements

7.1. Halo Exchange

The 'Halo Exchange' optimisation stems from enabling neighbour-to-neighbour communication of halo regions. Bordering rows and columns enable each worker to reduce communication overhead by lessening the need for synchronising with a central distributor every iteration. This decentralised approach allows for efficient data exchange, reducing latency and avoiding bottlenecks at a single point, thus improving the overall speed and scalability. This method allows each worker to operate more independently and in parallel, leading to a more efficient update of the game state, especially as the number of workers increases.

7.2. Parallel Distributed Implementation

Incorporating parallel workers within each AWS node leverages multi-threading, enabling each node to process multiple parts of the workload simultaneously. This potentially increases throughput and decreases processing time. By analysing the performance against the single-threaded implementation per node, one can measure improvements in terms of speedup and efficiency. As the complexity of the task increases with larger image sizes, performance can vary due to differing computational loads - a demonstration of performance gains via scalability and efficiency.

8. Conclusion

Our distributed implementation of the Game of Life demonstrates the seamless data transmission between the three main components ensuring efficiency in speed and scalability. We understand the bottlenecks that can occur from latency in connecting to AWS instances due to geographical scale. Additionally, multi-threading via parallel-distributed programming as well as leveraging efforts put in by each worker via the 'Halo Exchange' would be significant optimisations for our implementation.