Game of Life Coursework Report

Group Members: Jingxiang Zhang, Lingyi Lu

Part 1: Parallel Implementation

1.1 Functionality

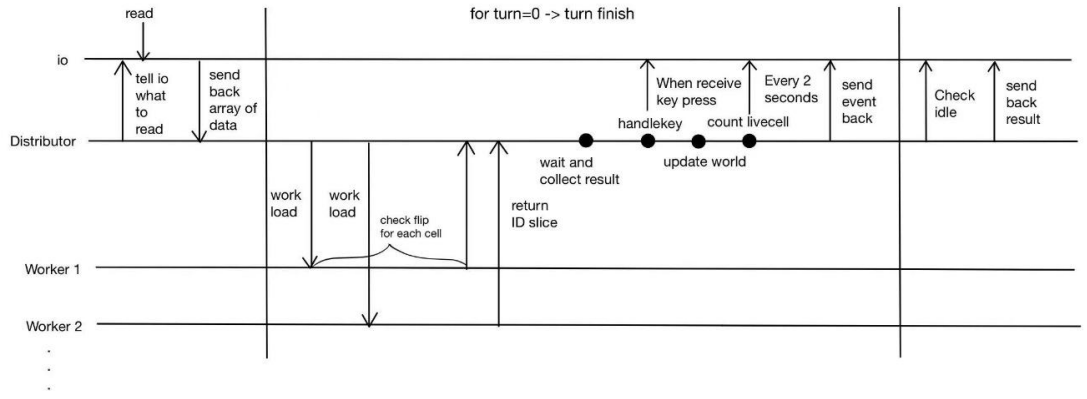
This design retains a single distributor that assigns work to a fixed pool of worker goroutines. Our parallel implementation achieved the features following:

1. The functionality builds on dynamic tracking of active cells: only cells that have changed state or whose neighbours might change are processed each turn.
2. The implementation uses flip detection to identify cells that become alive or die and maintains one lists of changed cell coordinates (nextList). This dynamic active-cell strategy avoids scanning the entire grid.
3. A ticker triggers periodic reporting of the current alive-cell count via AliveCellCount event every two seconds as specified.
4. At the end of the final turn the program sends a FinalTurnComplete event on the events channel with the list of living cells.

1.2 Final design

1.2.1 Final Design Specifications:

The distributor initiates an I/O request to retrieve an image and establish a data storage environment, including a 3 two-dimensional slice to store the whole world and the flip cell for each worker. 2 one-dimensional slices to facilitate subsequent cell calculations. The distributor then generates multiple goroutine threads. Within the main loop, the workload is distributed to each worker via a dedicated channel. The main loop will pause execution until all worker results have been collected. Each worker is responsible for calculating its assigned cells and returning a list of modified cells for the current iteration. The main loop subsequently integrates all cells from each worker into a one-dimensional array, identifies each cell's neighbors, and aggregates them into another one-dimensional array containing all cells designated for the next calculation cycle. Upon completion of these tasks, key presses, live cell counts, and any relevant SDL events are transmitted back to I/O. Once all iterations are complete, the distributor sends the final results back to I/O and terminates all communication channels.



1.2.2 Advantages:

1.I/O Optimization: The complete image array will be transmitted from the I/O instead of processing byte-by-byte.

2.Memory Efficiency: A single 2D slice will be utilised, eliminating the need for two.

3.Flip-cell: In each iteration, only modified cells will be recorded, as opposed to the entire image.

4.Neighbour Tracking (Highlight): For subsequent computations, only the neighbours of cells altered in the current iteration will be tracked. The subsequent iteration will exclusively process cells within this identified list.

5.No-Flip Short-Circuit: if an iteration produces zero flips, skip further computation (huge savings when the evolution stabilises early, e.g. 128×128×16000).

6.Resource Reuse: Workers and slices will be reutilised in each iteration.

7.Thread Safety: Neighbour counting will only be managed by the distributor to avoid data races.

8.Conditional Logic: if-statement are employed replace of modulo operations. The use of conditional statements is restricted to boundary cells; otherwise, direct calculation is performed.

9.Optimized Boundary Checks: Boundary checks will be executed only when a cell is situated at the boundary, as the majority of cells do not require an if-statement for this purpose.

10. Shared memory avoids channel copy and scheduling overhead, allowing workers to read/write the same data directly. Using lightweight mutex/condition synchronization offers higher performance and lower overhead compared to the channel-based version.

1.2.3 Disadvantage:

1.High fixed overhead at small thread counts: coordination (chunking, sending through channels, scheduling goroutines) adds significant per-thread overhead; when work per thread is small, these costs dominate. Therefore, more threads means that more coordination time will be
need. E.g.in 256x256x4000: only hundreds of cells flip in one turn. With the higher.

2.Sequential merge bottleneck: The main goroutine need merges all flips, runs getNeighbor, and updates Before sequentially to protect from data race.

1.2.4 Test results and benchmarks

All tests have passed with race flag enable. All benchmark tests are running in the same lab machine in one session to control variables. All benchmark tests are run 10 times to get mean value, and abnormal results will be ignored and retested.

- Figure 2 and 3 is different version testing on 512x512x1000
- Each version is based on the optimization of the previous version.

Version1: parallel workers with counting neighbour in every worker

Version2: Parallel workers with enhanced io

Version3: Parallel workers with counting flipped neighbour cell in each worker

Version4: Parallel workers with efficient use of slice

Final version: Parallel workers excludes “calcchunk” function and substitute mutex lock from IO channel.

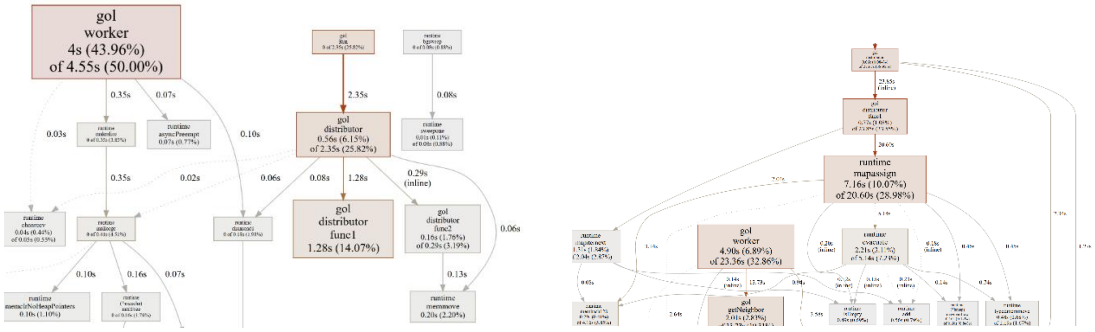


Figure 2: The pprof diagram for parallel design: Version3vsVersion4

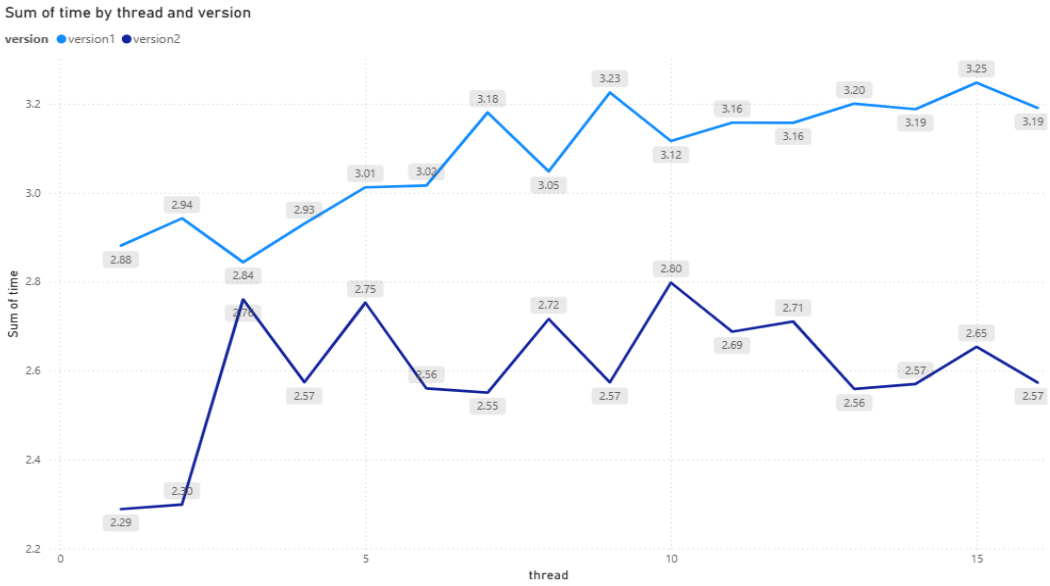


Figure 2: The time spent when 512x512x1000 for each thread in version1 & version2

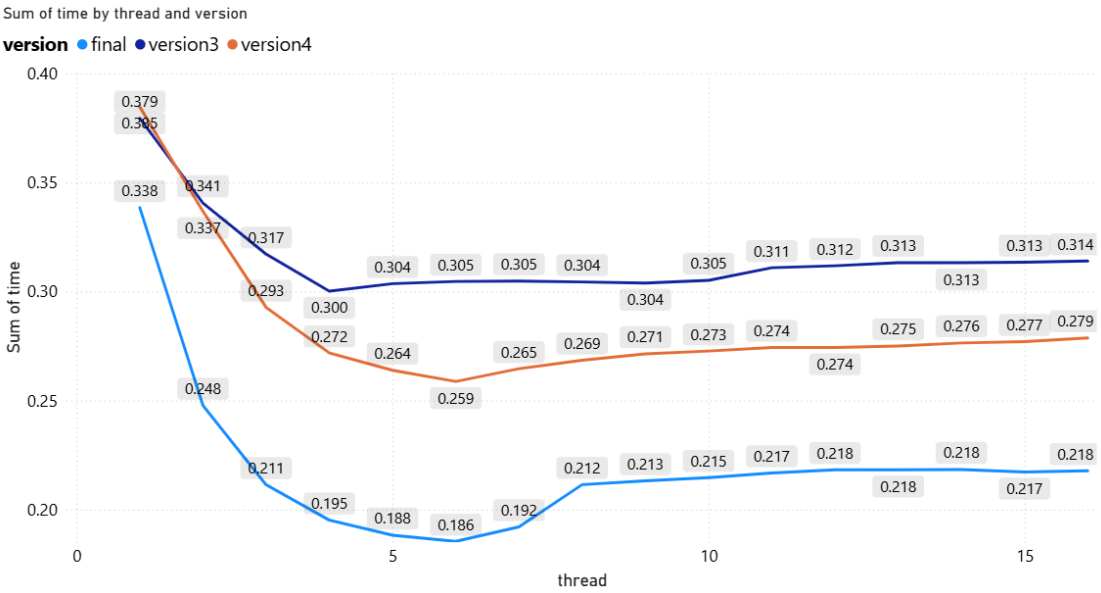


Figure 3: The time spent when 512x512x1000 for each thread in version3 & version4

Figure 4: The time spent on three different tasks with each thread in final design

Figure 4 presents an analysis of the computational time across three distinct grid sizes, with varying "flip cells" per turn, utilizing the optimized GOL design with different thread counts.

Explanation:
Conversely, the computational load per turn in the 256x256x4000 configuration is minimal, involving changes in only a few hundred cells. Increasing the number of goroutines in this scenario introduces a fixed overhead that diminishes efficiency. This overhead stems from broadcasting the "Before" grid to all workers, managing job and result transmission via channels, and subsequently merging flips and applying updates within the main goroutine. These coordination expenses surpass the minimal actual cell updates, leading to a performance degradation with more goroutines. Only larger grids, such as 512x512x1000 with thousands of flips per turn, provide sufficient work for each worker to offset this overhead. In the 128x128x16000 configuration, cell flips cease after turn 6. Consequently, our algorithm is designed to bypass further processing if no flips occur in a given turn, resulting in a total cost of less than 0.01 for the entire event.

Part 2: Distributed Implementation

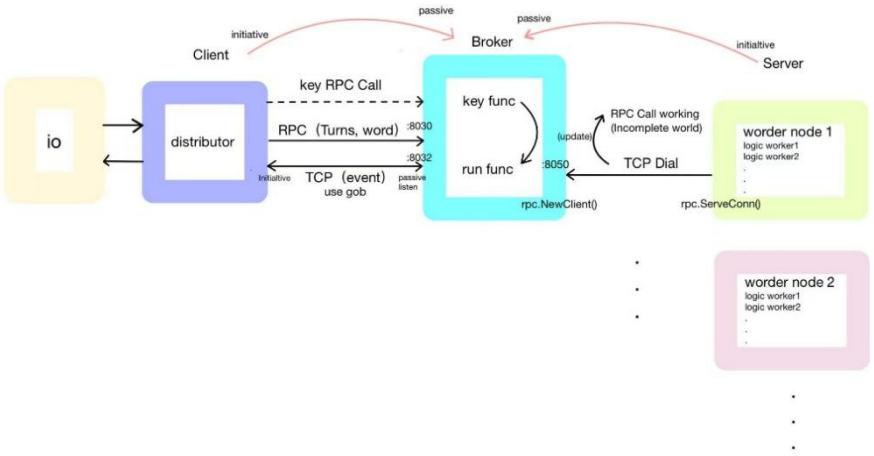
2.1 Functionality

We have done the parallel part first before we doing Distributed part, so distributed part idea is base on Parallel part. It support all functionality in parallel part, Include SDL live view.

2.2 Final design

Our final design centers on a single broker process with Go RPC that clients dial to start runs (Broker.Run), stream turn data, and inject control events (Broker.Handlekey). The broker tracks all worker servers (each on its own AWS node) via Broker.Register, partitions the board dynamically, and only ever initiates RPC calls outbound to those workers, so the worker fleet stays private yet elastic: any worker that dials in is enrolled, and dead entries are pruned mid-turn with work automatically reissued. Each worker in turn spins up its own pool of logic goroutines to execute the row ranges assigned by the broker, returning flip sets that the broker aggregates and relays to the client over a dedicated TCP stream. Keyboard events are buffered before runs start and funneled through a broker-managed channel while the run is live, allowing p,s,q… without disrupting the streaming connection . Because

every interaction happens through the broker's static IP (RPC for control, TCP for streaming).



2.3 Special design&Advantage

High Fault Tolerance

The broker tracks every worker via Broker.Register, pings them each turn, and reruns the current turn if any RPC call fails.

- Dead workers are removed from WorkerAddr and their rows are immediately redistributed, so a single node failure never crashes the job
- New add workers will be notice by broker but it won't affect current running test, it will apply to next test without crashing

Full World Transfer Only Once Between worker and broker

- The distributor pushes the full board to the broker only at the start of Broker.Run. From turn 2 onward, every RPC from the broker to workers includes just the current thisList (the complete set of candidate cells for that turn) plus it assigned dealing region with minimal halo rows they need for neighbor lookups, and every message back to t he client carries only CellsFlipped

Persistent TCP Stream

- Besides the RPC control channel, the broker keeps a long-lived TCP connection (streamListener) to the distributor for TURN/ALIVE/FINAL events. This avoids per-turn handshakes, lets AliveCellsCount updates go out every two seconds as required, and ensures keypress responses travel over the same reliable stream without loss.

Scalable Worker Pool

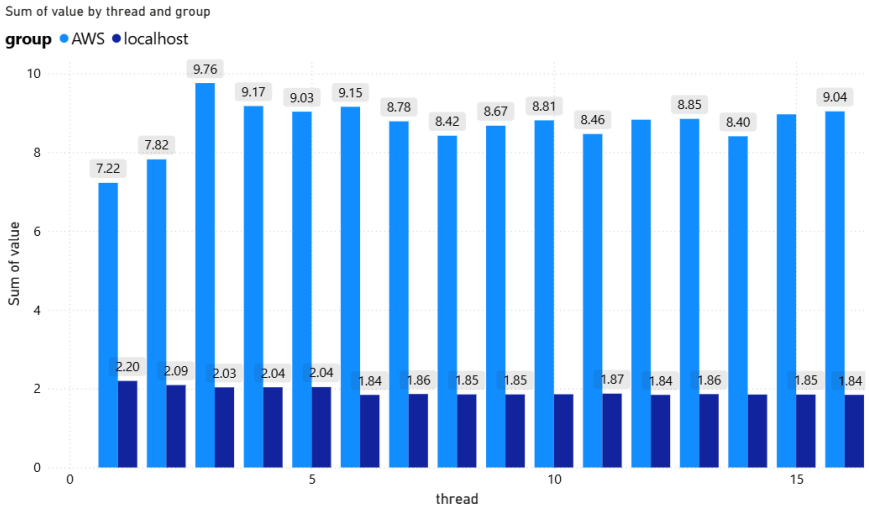
- The worker list is open-ended: any server that dials Broker.Register joins the pool, and the broker's allocatethreadtoworker routine repartitions threads every time a run starts or a worker disappears. Spinning up more AWS nodes automatically shortens each turn because the broker will spread rows across the larger pool.

One connection between Broker and server

- Because each worker dials the broker proactively, it doesn't need to expose any port; once the connection is established, keeping that single TCP link for all RPC traffic avoids repeated handshakes and saves both latency and resources.

Test Result and Benchmark

All tests successfully passed with the race detector enabled.



Testing was conducted independently on both a Lab machine local host and an AWS t3.micro instance. The distributed system demonstrated significantly slower performance compared to the local system, primarily due to:

- The overhead associated with remote communication, which exceeded the computational advantages of parallelism.
- The substantial impact of network instability on benchmark performance.

These findings underscore a critical insight in systems engineering: distributed computing does not inherently guarantee faster performance. Rather, performance is heavily influenced by network overhead and the granularity of the workload.

3. Conclusion

This course gives students a good opportunity to understand how concurrency and distributed systems behave in practice, beyond their theoretical advantages. Through implementing both the parallel and distributed versions of the Game of Life, we learned that performance is determined not only by algorithms, but also by coordination overhead, workload size, and the cost of communication.

Our parallel design achieved strong results on larger grids, where dynamic flip-cell tracking and efficient neighbour propagation provided enough computation to offset goroutine and merging overheads. However, when the number of active cells per turn was small, the fixed coordination cost limited scalability.

The distributed implementation highlighted even clearer trade-offs. Although our broker–worker architecture was fault-tolerant and scalable, remote execution on AWS was significantly slower than local execution due to RPC latency, network variability, and low per-turn workload.

Overall, the coursework demonstrated that effective system design requires balancing parallelism, communication, and computational granularity.