Conway's Game of Life: A Quantitative Evaluation*

Matvii Ustich, Mihály Tóth-Tarsoly and Tom Lam

School of Computer Science, University of Bristol
{oa24447,wq24423,ye24597}@bristol.ac.uk

Abstract

Conway's Game of Life is a familiar optimisation problem involving the space- and time-efficient computation of iterations of a cellular automaton. We implement Conway's Game of Life using serial, parallel, GPU, and distributed approaches. Optimisations such as lookup tables and bit-packing significantly reduce CPU overhead, with Rust outperforming Go, and a CUDA bit-packed kernel reaches near state-of-the-art performance. The distributed architecture remains fault-tolerant under node failure and addition. Deployment is automated using Terraform for seamless cloud scaling. The implementation of a halo-exchange scheme in Rust further improves scalability by minimising communication. Analysis of results shows bit-packing and parallelism are the dominant performance factors.

1 Introduction

Conway's Game of Life is a cellular automaton that uses a world w of $n \times m$ pixels wrapping around at the edges, where each cell can be either alive (1) or dead (0). The evolution for each cell $w_t(x, y)$ in current turn t with $x \in [0, m-1]$ and $y \in [0, n-1]$, is calculated based on its neighbours, according to the following rules, where $n_t(x, y)$ represents the count of alive neighbours surrounding cell (x, y):

$$w_{t+1}(x,y) = [n_t(x,y) = 3] \\ \vee \ \big(w_t(x,y) \wedge [n_t(x,y) = 2]\big) \tag{1}$$

This evolutionary algorithm presents an interesting problem when considering parallel or distributed computations of iterations of the automaton, because cells are dependent on their neighbours. This restriction means that we cannot simply just break up a grid into sections naively, because each individual process must also have access to the edge cells of its neighbouring grids. In addition, these edge cells cannot just be communicated once, because they may change as the simulation progresses.

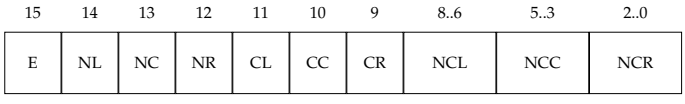
2 Basic serial implementation

Perhaps the simplest way to implement the Game of Life is by using a basic serial implementation, where each cell is evolved individually in sequence. One complication that must be dealt with is that cells on the edges have their neighbours on the other side of the board.

A modulo-based method can be used to calculate the coordinates of the neighbouring cells in order to look up their values. In addition, instead of counting neighbours directly, we represented cells as either 0 or 1, allowing a simple sum to be calculated.

2.1 A lookup table

A more unique approach for calculating the next state for given cells is by using a lookup table. An inspiration came from reading chapters 17 and 18 from Abrash's Book [1]. It led to an interesting idea – by storing some metadata about the given cells, we can deterministically compute their next state without having to look at the neighbouring cells, which should reduce the computational overhead. First, we change the board structure so that cells are stored in *triplets*. Every triplet has enough space to store the current and the next states of three horizontally adjacent cells and the external neighbour count for the current generation. Overall, the structure can be represented as follows:



E: 0 if cell is internal (no wrapping), 1 if an edge
NL, NC, NR: next life/death states for the left, centre and right cells respectively
CL, CC, CR: current life/death states for the left, centre and right cells respectively
NCL: external neighbour count for the left cell
NCC: external neighbour count for the centre cell
NCR: external neighbour count for the right cell

Figure 1: *Triplet layout.*

Although every cell can have up to 8 alive neighbours, only 3 bits are dedicated to every external neighbour count. This is because by summing up one of the external neighbour counts with the right cell (e.g. `NCL` with `NC`), the total neighbour count can be found. This means that one triplet contains enough information to compute the next states for the cells stored inside it. Therefore, a lookup table of size 2^{16} can be constructed to find the next state of the bits described by a triplet (i.e., `table[triplet]=NL, NC, NR`).

Once we have the lookup table and the world represented by the list triplets, the next state of the world can be found in two stages:

1. Iterate through the list of triplets, assigning the correct values to `NL`, `NC` and `NR` using the lookup table.
2. Iterate through the updated list of triplets. For every cell that has a different next state compared to the current state, update external triplets' neighbour counts. Make current state equal to the next state.

The above allows us to advance the board state without necessarily having to address every neighbour for every cell, which should reduce computational overhead. The difference in performance associated with this change can be seen in Figure 2.

*Presented as a report for Computer Systems A

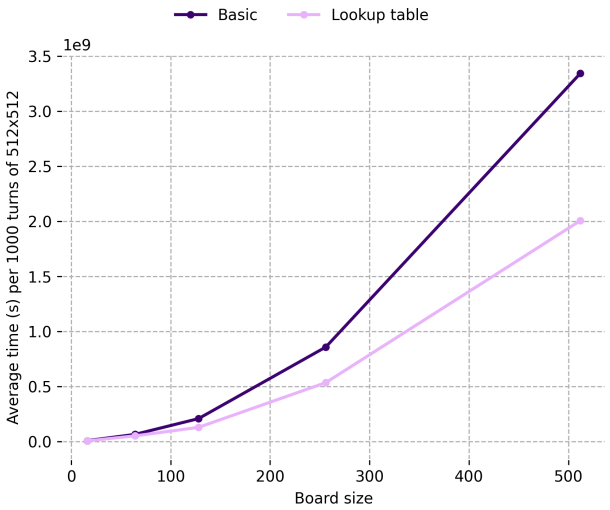


Figure 2: *Time to complete 1000 iterations using the serial implementation as the board size changes.*

3 Parallel implementation

The first concurrent implementation attempted was one that operates in parallel, across multiple worker routines. We initially implemented this in the Go programming language.[1]

3.1 A simple optimisation

An immediate possible optimisation is to replace the modulus operation, which is computationally expensive, with a counter that rolls over automatically. Interestingly, we didn't notice an increase in performance when we attempted this change. To investigate this, we used the *pprof* tool, and we noticed that a very large amount of time was being spent on incrementing the for loop counter. When we looked at the generated assembly, we found that Go's compiler didn't unroll the for loop, which could be why it didn't result in an increase in performance. The profiling results can be seen in Figure 3.

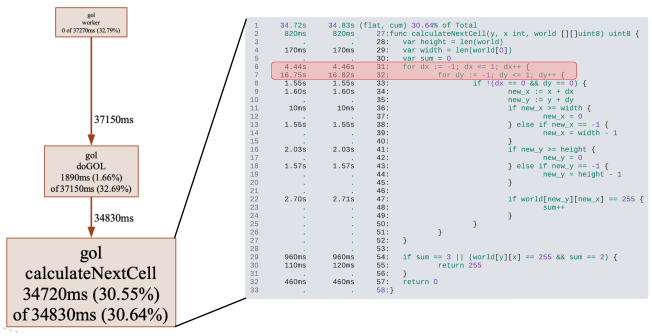


Figure 3: *Profiling results of the initial, basic parallel implementation.*

[1] Go implements concurrency using *goroutines*, lightweight threads that are managed by the Go runtime. They are conceptually separate from operating system threads: multiple goroutines may operate in the same OS thread.

3.2 Splitting up the board

When splitting up the board, it is important to ensure that the board is split up in a way that maximises the utilisation of each worker. This means that we keep track of the number of remainder board rows, and split them evenly across the workers. A previous implementation used a simpler algorithm that just assigned the remainder as one big job. Interestingly, this change did not result in a large change, as the results show.

3.3 Bit-packing with batch cell state computation

Since the state of any given cell is either dead or alive, one cell can be represented by a single bit. This leads to the idea that we can use an n-bit word to represent n consecutive cells. In usual systems, the default size of a single word is 64 bits; therefore, we can encode 64 cells in a single word, drastically reducing memory usage compared to the basic implementation. Because computers operate on whole words, this hints that it is possible to update states of several cells simultaneously without using concurrency. Studying a paper discussing bitwise parallel bulk computation [2] proved that we can update all cells in a single world without having to treat each one of them individually. Assuming the operating system is little-endian and the least significant bit corresponds to the horizontally left-most cell, the algorithm 1 can be used to compute the next states for 64 cells simultaneously. First, the algorithm finds all neighbours as shown in 4. Second, it computes the neighbour count for every cell.

`ind - wpr - 1`	`ind - wpr`	`ind - wpr + 1`
`ind - 1`	`ind`	`ind + 1`
`ind + wpr - 1`	`ind + wpr`	`ind + wpr + 1`

Figure 4: *Words layout and their indices in the cells list;* `ind` *is an index of the word in the middle and* `wpr` *stands for words per row*

However, since neighbour count is in the range $[0, 8]$ and four bits are required to store it, we can't use simple addition. Instead, we do the summation by creating four separate words, which will then give us enough information to be able to tell the neighbour counts for every cell deterministically. We use half- and full-adders to calculate `w, x, y, z`, which are then used as follows:

1. `w` is used to make the cell alive if it has an odd number of neighbours
2. `y, z` are used to keep the cell alive only if it has 2, 3, 6, 7 neighbours
3. `x` is used to make sure that no cell with more than four neighbours survives

All of the conditions above applied in order to ensure that the Game of Life rules are satisfied.

Algorithm 1 Bit packing with batch computation

```
1: procedure NextWord(cells, ind, wordsPerRow)
2:     // Find neighbours
3:     top ← cells[ind − wordsPerRow]
4:     bottom ← cells[ind + wordsPerRow]
5:     left ← (cells[ind] ≪ 1) or (cells[ind − 1] ≫ 63)
6:     right ← (cells[ind] ≫ 1) or (cells[ind + 1] ≪ 63)
7:     topLeft ← (cells[ind − wordsPerRow] ≪ 1)
           or (cells[ind − wordsPerRow − 1] ≫ 63)
8:     topRight ← (cells[ind − wordsPerRow] ≫ 1)
           or (cells[ind − wordsPerRow + 1] ≪ 63)
9:     bottomLeft ← (cells[ind + wordsPerRow] ≪ 1)
           or (cells[ind + wordsPerRow − 1] ≫ 63)
10:    bottomRight ← (cells[ind + wordsPerRow] ≫ 1)
           or (cells[ind + wordsPerRow + 1] ≪ 63)
11:    // Stage 0
12:    (l, i) ← FullAdder(top, bottom, left)
13:    (m, j) ← FullAdder(right, topLeft, topRight)
14:    (n, k) ← HalfAdder(bottomLeft, bottomRight)
15:    // Stage 1
16:    (y, w) ← FullAdder(i, j, k)
17:    (x, z) ← FullAdder(l, m, n)
18:    // New word computation
19:    newWord ← cells[ind]
20:    newWord ← newWord or w
21:    newWord ← newWord and (y xor z)
22:    newWord ← newWord and not x
23:    return newWord
24: end procedure
```

3.4 Results

Benchmarking this implementation, we can see that the speed scales with the number of workers in Figure 5. However, the speedup slows down and eventually stops altogether. This is a common theme with all of our concurrent implementations. We believe this is caused by the limit imposed by the number of physical cores in the computer. In addition, as the board is split up more, there is a greater overhead involved in splitting and reassembling the world, as well as coordinating the threads.

3.5 Rust

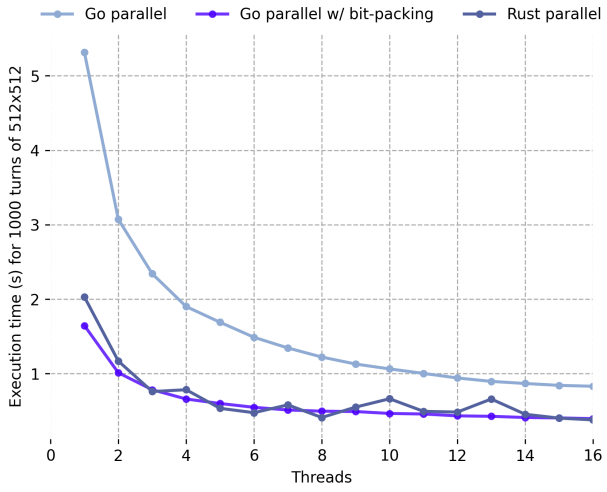


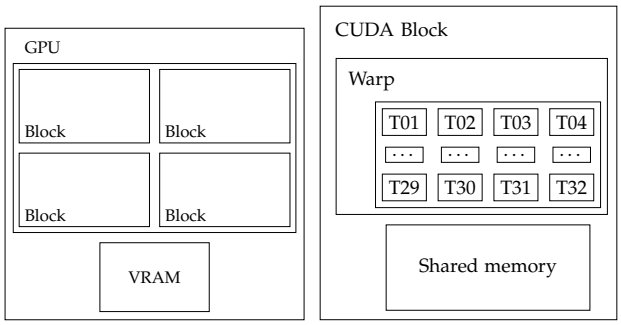
Figure 5: *Time to complete 1000 iterations of a 512x512 board for the Rust parallel implementation, compared with other implementations.*

We also implemented a parallel implementation using Rust. A work-stealing thread pool was used from *Rayon*, a data parallelism library. In order to satisfy Rust's memory safety constraints, we break up the updated world into several disparate slices which are handed out to the threads. In addition, another optimisation used was to represent the world using a single one-dimensional array rather than several nested two-dimensional arrays, which should theoretically perform better because the CPU does not have to follow two pointers to reach the data, and hence potentially improving caching.

We found that the basic Rust implementation was much faster than the basic Go implementation, and even sometimes faster than the best Go bitpacking implementation. It is likely that this is because of Rust's use of manual memory management instead of garbage collection.

4 CUDA implementation

4.1 CUDA programming model



(a) *GPU model* **(b)** *A CUDA block*

Figure 6: *CUDA programming model of a GPU*

The GPU is specialized for parallel computation given its higher memory bandwidth and density of transistors for data processing rather than control logic, unlike a CPU. It consists of an array of Streaming Multiprocessors (SM) that execute warps of threads in parallel with SIMD-like instructions. The work is scheduled on the SM as blocks which are independent from each other. Each block consists of its own shared memory on the chip which provides a much highly access rate compared to the off-chip global memory (VRAM).

4.2 Basic implementation

We defined the size for each block to be 32×8, which means each block consists of 256 threads. For the kernel launch in each thread, it fetches and sums its 8 neighbours from the global memory in VRAM and applies the life rule to compute the next state for the cells. To integrate the CUDA kernel with the Go SDL window, we used CGO[2] to create bindings to the external CUDA functions. We only allocate CUDA memory once, at the start of the simulation in Go, and free the memory when the controller is terminated.

[2] An interface layer to call C code from within Go.

4.3 Bit-packing implementation

We implemented the same algorithm 1 used for Go parallel. Having experimented with different memory block sizes, we converged on a configuration of 8×8. Although large block sizes (e.g. 32×8) offer better memory coalescing, allowing warps[3] to read wider consecutive words from the global memory, our kernel is potentially dominated by arithmetic operations, large amount of registers per thread and branch divergence caused by separately handling boundary words. Smaller blocks enable the GPU to schedule more blocks on the SM concurrently, hence allowing more warps to be executed simultaneously, hiding the latency from both memory access and branch divergence.

4.4 Result evaluation

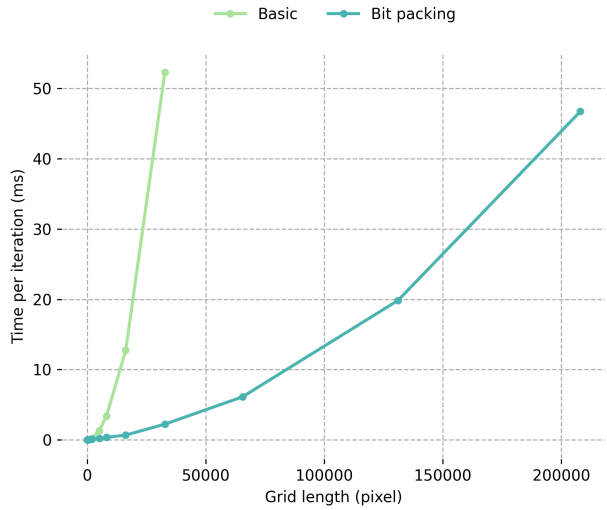


Figure 7: *Average time for each iteration with varying number of grid pixels in CUDA*

Figure 8: *Average time to complete 1000 turns of 512x512 in Go with SDL window rendering for various CUDA implementations*

To compute the raw performance of our CUDA kernels and compare them against the state of the art approaches in Table 1, we only retrieve the last turn result from GPU after all iterations are finished, eliminating the performance deterioration caused by memory allocations and copies. Our bit-packing and basic algorithm achieve remarkable results of 9.344×10^{11} and 2.077×10^{10} cell updates per second respectively. However, it could still be further optimized by using a multi-step kernel, performing multiple steps of the simulation from a single inital load from the global memory to reduce memory traffic, using assembly instructions to replace some of the arithmetic operations in the next state calculation, and the use of a wrap shuffle to share vertical halos without accessing shared or global memory.

[3] A group of 32 threads that are executed simultaneouly with SIMT.

Our basic (RTX A2000)	**Our bit-packing**	**Fujita et al.** [2] (GTX Titan X)	**Dijkstra et al.** [3] (GTX 1080 Ti)
2.077×10^{10}	9.344×10^{11}	1.990×10^{12}	5.389×10^{12}

Table 1: *Maximum measured performance (cell updates/s) of the state-of-the-art CUDA-based GoL implementations*

5 Distributed implementation

We also attempted an implementation that works across computers using a distributed architecture. The system, as shown in Figure 9, was split up into separate programs with a separation of concerns; a controller, which acts as a client; a broker, which co-ordinates the workers and breaks up and reassembles results; and workers, which do the actual computation. Communication between these components was implemented using RPC calls.

Worker discovery was implemented using the publish-subscribe model. This was to ensure maximum flexibility, and to remove the requirement for the broker to know the IP addresses of each individual worker. This also simplifies deployment to the cloud.

In this way, when the worker starts up, it connects to the broker and sends a subscription request with an RPC call. The broker accepts the worker onto its list of subscribers, and opens an RPC client connection. This gives great flexibility in dealing with workers individually, and allows implementation of fault tolerance.

In order to deal with the halo problem, a so-called *coupled halo exchange* was implemented. Here, the broker sends the additional halos required for each job to be completed to the workers.

Benchmarking the basic distributed implementation As expected, the distributed implementation scales well with the number of workers added, up to a point. However, sending the board back and forth every turn has high bandwidth requirements, and slows down the simulation.

5.1 Parallel workers

We also implemented parallel processing of the world slice for our bit-packing implementation on CPU, using the techniques previously discussed. Due to the limited number of threads available on a single computer, this implementation cannot be run properly locally, and requires the deployment of workers in the cloud. We were unable to run instances with enough vCPUs to be able to collect meaningful data,

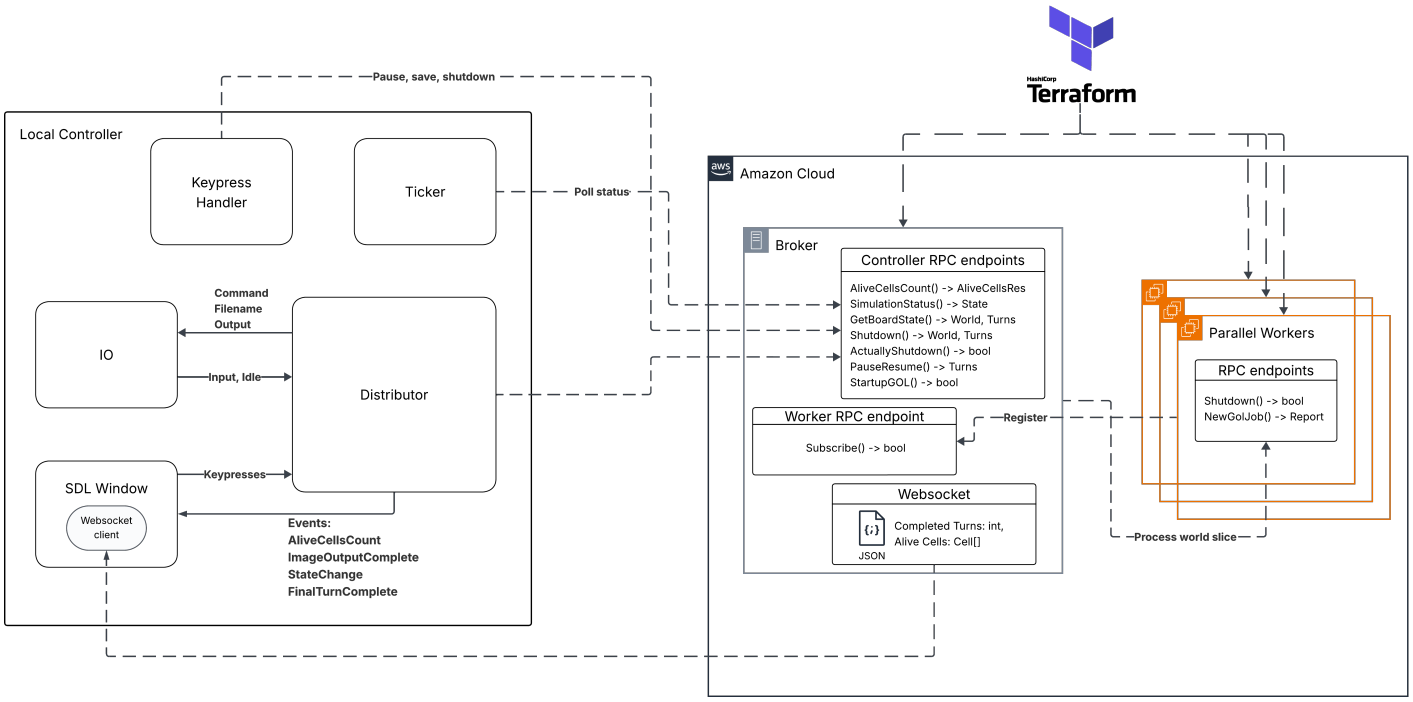


Figure 9: *The Go distributed architecture*

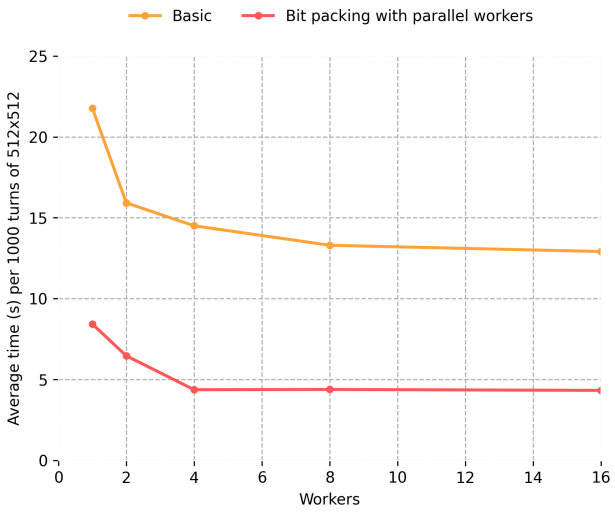


Figure 10: *Average time (sec) per 1000 turns of 512x512 for various distributed implementations on AWS*

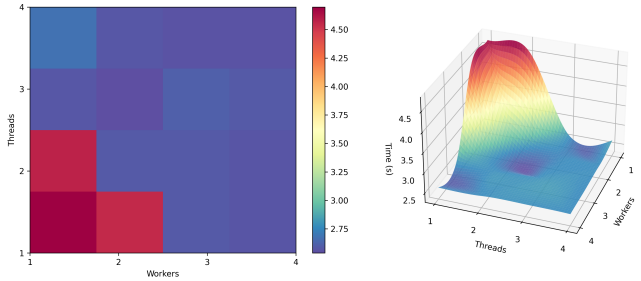


Figure 11: *Average time (sec) per 1000 turns of 512x512 board for different number of threads and workers for bit packing distributed implementation running on M3 Pro chip*

as the limit was set at 1vCPU per instance, making parallel workers meaningless. When it was tested locally, it resulted in a performance improvement. This can be seen in Figure 11

Implementation	Time (s)
Bit-packing	324.8
Parallel worker	325.9

Table 2: *Performance comparison of parallel worker, with 4 workers each with 4 threads, and 4 bit-packing workers, on a* 5120×5120 *board for 1000 turns.*

5.2 Fault tolerance

Our current Go implementation of the distributed game of life has reasonably robust fault tolerance. Our worker pool works using a publish-subscribe model where the workers will each call the subscribe RPC endpoint on the broker to register its IP address at the start. If a worker disconnects from the simulation while it is running, the broker will put the job back to the jobs channel and wait for another available worker to acquire the job. If there are no workers in the worker pool while the simulation is running, the simulation will come to a halt but the simulation state (running, paused) is still preserved, keypresses will still work and the simulation will resume from the state where it is left off after a new worker is connected.

5.3 SDL Live View

To return each completed turn and its world state to the controller for visualisation, we implemented a WebSocket connection between the broker and the local controller in Figure 12. Compared to RPC, WebSockets have a smaller message overhead, helping to minimise latency. Because WebSockets

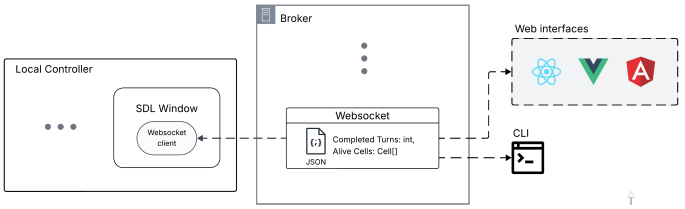


Figure 12: *The proposed multi-viewer design for the game of life simulation*

run over TCP and support sending JSON-encoded objects, they allow efficient transfer of the AliveCells list. They also support multiple simultaneous client connections, enabling multiple views and even potentially web-based interfaces in the future.

5.4 Rust

As an extension to the main task, we also wrote a version of the distributed implementation in Rust. As simple RPC calls were not available, the *gRPC* framework was chosen to communicate between the programs. This involved writing a Protobuf definition for the services and their behaviour.

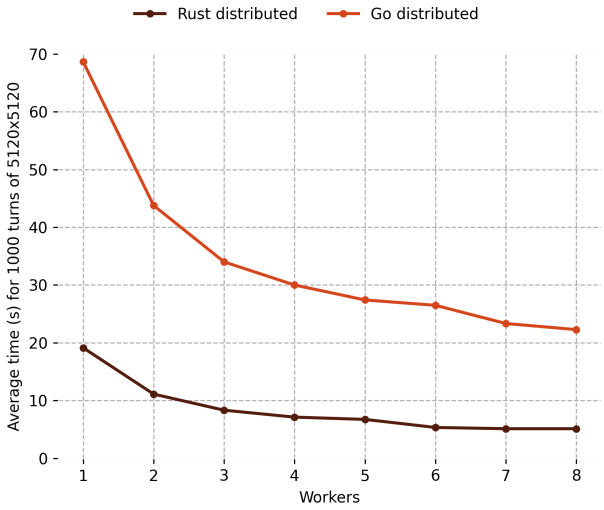


Figure 13: *Performance comparison between Rust distributed system with Halo Exchange and the Go distributed system*

Halo Exchange One of the biggest differences between the Go implementation and the Rust implementation is that the Rust implementation uses a full halo exchange scheme. This solves the halo problem in an elegant way; rather than having to send back the board section to the broker after each iteration and reassemble the board before sending it again for the next turn, the workers instead communicate with each other, in a peer-to-peer architecture. The role of the broker shifts instead to a much more supervisory role; it sends a synchronising pulse to signal a turn should start, and splits up the board initially. Other than this, the workers operate entirely independently.

On the worker, when a turn is signalled, it sends a request to the workers holding the halos that it requires to complete a turn. Once these are received, it can compute the updated board state. Before this can be persisted however, it must wait until it has already received halo requests from the other workers. This is to ensure that it doesn't continue before the other workers have received the initial halo. When a board state update is required by the controller, the turn signalling is paused, and the broker requests the board state using a separate RPC call.

This architecture greatly reduces the data transfer occurring as only the halos have to be exchanged. This increases performance, especially for larger boards, where the amount of data being transferred every turn becomes increasingly significant. A diagram showing the architecture can be seen in Figure 14.

Benchmarking the halo exchange We benchmarked the Rust halo exchange implementation against the basic distributed implementation with no SDL, bit packing or parallel workers so that they could be fairly compared. As seen in Figure 13, we found that the halo exchange scheme has far greater performance, with significantly reduced turn times. The use of a 5120×5120 board was required; the 512×512 board was completed too quickly by the Rust implementation, and the ticker was too slow to capture differences.

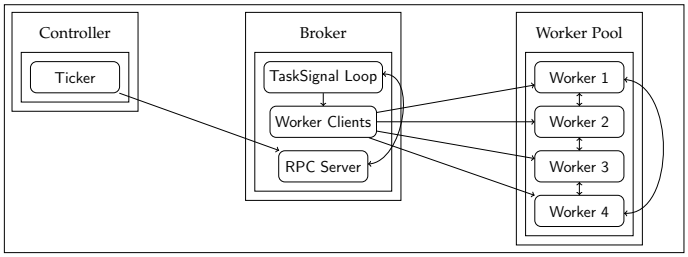


Figure 14: *Diagram of the Rust distributed architecture using a Halo Exchange scheme.*

5.5 Scalability

In our Go distributed implementation, we used Terraform as our infrastructure management framework to automate the deployment of AWS nodes using shell scripts, as shown in Figure 9. Each instance is launched from a custom AMI with all dependencies preconfigured, pulls the latest version of the repository, and starts the server to establish connections between the broker and workers. This approach greatly improved our efficiency and allowed us to benchmark using arbitrarily large numbers of instances.

References

[1] Michael Abrash. *Michael Abrash's Graphics Programming Black Book, Special Edition*. Coriolis Group, 1997.

[2] Toru Fujita, Koji Nakano, and Yasuaki Ito. "Fast Simulation of Conway's Game of Life Using Bitwise Parallel Bulk Computation on a GPU". In: *Int. J. Found. Comput. Sci.* 27 (2016), pp. 981–. URL: `https://api.semanticscholar.org/CorpusID:32397603`.

[3] Julia Dijkstra and Jonathan Brouwer. *Game of Life: How a nerdsnipe led to a fast implementation of Game of Life*. Accessed: 2025-11-16. June 2023. URL: `https://binary-banter.github.io/game-of-life/`.