Parallel and Distributed Implementations of Conway's Game of Life

PETER LILLISTONE (JR22741), FINN COOPER (SD22404)

Compiled November 2, 2025

1. INTRODUCTION

This report covers multiple implementations of Conway's Game of Life in Go, using both parallel and distributed design patterns. It offers brief descriptions of the approaches taken to program design and analysis of each implementation.

2. PARALLEL

A. Basic Implementation

Starting with a simple serial implementation of the Game of Life, our first step was to divide the work between a number of worker goroutines equal to the value passed in by the 'Threads' flag. Our solution uses both shared memory and channels to handle the communication between the workers and the distributor; each worker is passed a copy of the whole Game of Life world, and given a start and end row to work between. We chose to pass the whole world to each worker in this way so that each worker has easy access to the halo regions surrounding its target segment to check for neighbour aliveness. The use of channels also makes it easy to make sure all workers are finished and collate the segments in the distributor, before moving on to the next turn.

The 'segment height' is used to give the start and end rows for all but the last worker, and is calculated as follows:

$$S = floor(imageHeight / Threads) \tag{1}$$

The last worker computes the remaining rows, ensuring that if an image is given with height that is not divisible by the number of workers, it still handles the entire world.

Each worker then uses the Game of Life rules to calculate the state of the cells for the next turn and input them into a new 2D slice that has height equal to the segment height. Once complete, these are sent by value along the worker's return channel, and are assembled by the distributor.

We considered optimising this approach to only pass the desired segment plus the extra row above and below to each worker. This would have decreased the amount of memory used by each worker, as well as marginally reducing the time taken to process each turn. Instead we decided to focus our attention on a new design described in the consumer model section.

B. Consumer Model

Expanding on our basic parallel system, we experimented with an approach that relies on shared memory; two worlds are stored in the distributor (`readWorld` and `writeWorld`), such that the workers can write to one world without disturbing any reads due to user input or polling the number of live cells. Each worker is passed pointers to both `readWorld` and `writeWorld`. In this way, workers are 'consuming' data from `readWorld`, processing it, then writing to `writeWorld`.

We started with a 2D slice of booleans (`checkedWorld`) which represented whether or not each pixel of the image has had its neighbours checked by a worker. To avoid data races, a mutex lock on the distributor was used so that only one worker could update the 2D slice at a time. However, due to the number of mutex lock and unlock requests, and the relatively short amount of time for the workers to process a single pixel's neighbours, the system relied too heavily on the mutex lock. This caused slowdowns while the workers were waiting on the lock, such that this implementation was considerably slower than our basic implementation.

In light of this, we adjusted our method so that workers are checking for rows at a time rather than individual pixels. We ended up with an array of booleans (`checkedRow`) stored on the distributor which represents rows of the input image. When trying to look for available work, workers will have to claim the lock to then check whether the row they attempt to process has already been worked on, indicated in `checkedRow`.

With this `checkedRow` array, the workers are not confined to a particular segment of the image. If one worker is slowing down, another can pick up the slack by consuming more rows nearby. In the name of efficiency, we give the workers a start point that is evenly distributed throughout the image, so that on average the workers will all finish at the same time.

One issue with this is that if the height of the image is smaller than the number of threads, then the remaining workers are not given any work, and wait idly. This is also the case with the basic implementation, so we made the reasonable assumption that image height would always be greater than the number of threads.

C. Analysis

Our data from these two parallel GoL implementations was obtained by running a benchmark that saves the total time processing a 512x512 image for 1000 turns, with different thread counts. As can be seen in Fig. 1. the consumer model is faster than the basic implementation for all thread counts.

Our basic implementation (*Parallel 1*) scales somewhat evenly with the number of threads assigned, flattening out at about 9-10 worker threads. This is expected as the system running these tests has 12 logical cores, and will begin to balance the threads

across multiple cores at 10 workers + distributor + benchmark process, meaning no noticeable improvement as the thread count increases.

In contrast, the speed increase per thread flattens out abruptly at 4 threads for the consumer implementation (*Parallel 2*), despite still being faster for all thread counts; this is likely because of the way the workers finish their execution. Crucially, when a worker is out of work, it checks each row of the image incrementally to see if it has already been processed by other workers, and only exits when loops around the whole image and returns to its starting row. This enabled the workers to pick up each others' slack if the load between them was uneven, but results in wasteful processing at higher thread counts, when the workers quickly finish processing the image, but still check all other rows until they get back to the start.

One possible solution to this problem is to have a flag of some kind that activates when all rows have been processed, stopping all workers immediately. This could be a channel, or a variable in the distributor that is passed by reference to each of the workers.

Fig. 1.

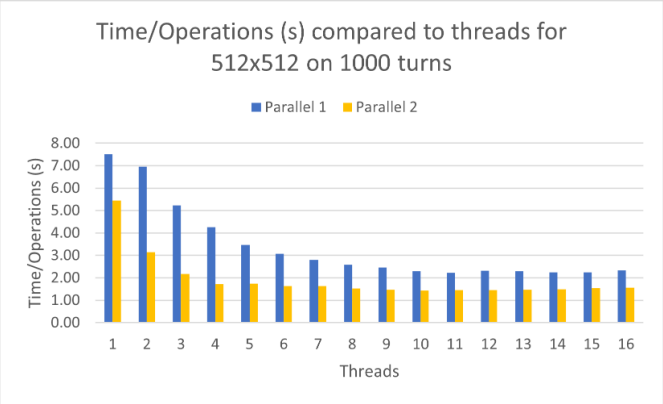

Fig. 1. Graph showing threads against time to run 512x512 image for 1000 turns

Fig. 2. shows the trace of CPU usage of the basic parallel implementation when running with 8 threads on a 512x512 world. This shows that the processes are being split properly across all the available processors and also that 9 threads are running continuously throughout execution.

Fig. 2. Go Trace of basic parallel implementation run with 8 threads

3. DISTRIBUTED

A. Basic Implementation

Our first step in developing our distributed implementation was splitting our Game of Life logic from our distributor. We approached this by using RPC connections between the client and worker, following the subscriber-publisher model. The main procedure exposed through this method is the serial implementation of our Game of Life logic, at its simplest it carries out a set number of turns on a given starting board state and then returns the final state of the board after computation is complete. After this we had to implement I/O functionality between the client and server to allow for handling intermediary processes whilst the main procedure was running. For this we used server-side channels inside the RPC structure to allow other RPCs to interact with the main procedure, this allowed us to send an interrupt command into the main procedure, which is received between turns, handled, and then the next turn commences.

From here we moved onto supporting multiple workers, in order to achieve this we again needed to split our structure, this time we added a broker server to act as a middle-man between the client and the workers. This broker server also uses RPCs to take the entire state of the world and split it into smaller slices, which it sends to individual workers. It also acts as a forwarder for the other RPC actions such as pausing and outputting. Upon these workers finishing, the broker receives each of their individual final states and then reconstructs the final world to send back to the client.

This, however, leads to a problem; due to the way the Game of Life functions, individual segments of the world require some context about the other segments in order to compute their next state, notably the adjoining rows of the adjacent world segments. In the parallel implementation this could be solved using shared memory, but as we are now assuming our workers are physically separated we can't use the same technique. Our chosen resolution to this problem was to use a halo exchange system.

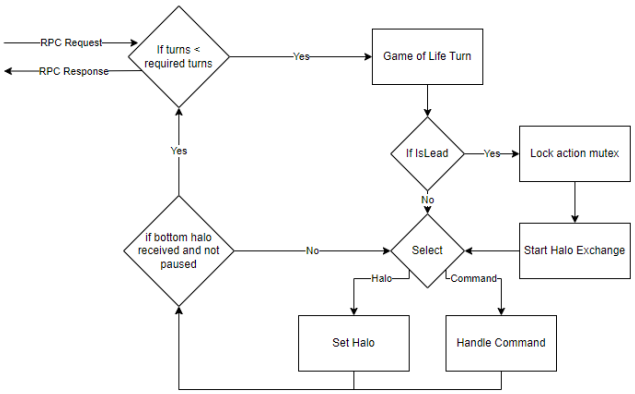

Fig. 3. Flow diagram describing halo exchange process

B. Halo Exchange

In order to maintain consistency, our halo exchange process is also implemented using RPCs. More specifically, it uses a cyclic system of RPCs in which one of the workers is elected as the leader by broker. This leader is in charge of starting the halo exchange process and locking the action mutex lock, which halts client inputs until the process is complete. The process itself involves workers dialing their successive worker (starting with the lead worker) with their bottom halo contained in a request.

The receiving worker will set this in their game state and dial the next worker. Once the lead worker is dialed again it will break the cycle by sending its top row as an RPC response, this then propagates back through the workers leaving them with their updated adjacent rows. Once the lead worker has received its top row it unlocks the action mutex and the next turn of the Game of Life can be computed.

C. Parallelising Workers

Now that we had a distributed system implemented, our next goal was to combine our parallel implementation with the individual distributed workers. This process was relatively simple as it just required adjusting the internal logic of the main RPC. From there we were able to run a similar benchmarking process to the parallel implementation, where we plotted the time/operations against the number of threads used per worker.

Fig. 4. contains our data collected from running the parallel distributed system locally with 4 workers and 1 broker, and also shows the results from running the parallel consumer implementation for comparison. As shown, our parallel distributed does follow a similar exponential decay as the parallel, even though it is far less pronounced. It seems to reach terminal speed around the same number of threads as the parallel. However, the important difference to note here is that this graph plots threads per worker, as the parallel implementation only has one worker it is actually running a quarter of the number of threads in total compared to the distributed system.

This is likely why the distributed reaches terminal speed around 4 threads per worker, as this means that 16 active worker threads are trying to run simultaneously. However, the local machine running these only has 12 logical cores, and as such between 3 and 4 threads per worker it has to begin process swapping in order to run all the threads. Whilst this process makes it appear to be running all the threads simultaneously, it is in fact still only running 12 threads at any one time and as such there is little room for speed improvements beyond this point, as shown by the levelling out for threads counts above 4.

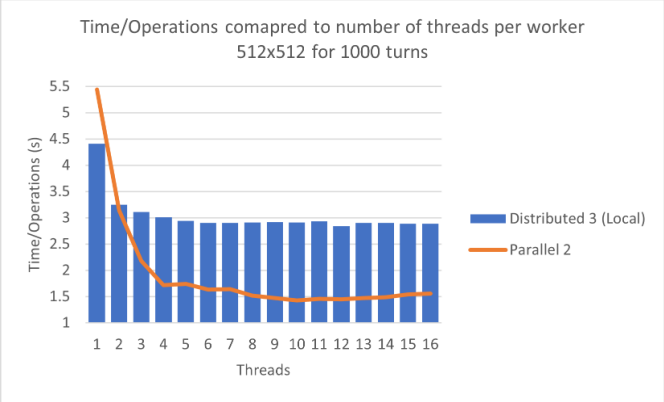

Fig. 4. Graph showing operation time decreasing with higher thread count for each worker

C.1. Local vs AWS

After testing the distributed parallel locally we decided to benchmark the same code in an actual distributed system, for this we used four t2.large AWS instances for the individual workers, one t2.micro AWS instance for the broker, and the same local machine with 12 logical cores for the client. Our expectation was that the AWS version would be significantly faster, especially for larger thread counts, as unlike the local version each worker would be running on their own separate system with dedicated vCPUs. The result of this are shown in Fig. 5. and they show the opposite of our assumption: the AWS-deployed system was considerably slower even at higher thread counts. There is also no exponential decay in the AWS results which would usually be expected from increasing the thread count.

At a first glance it might seem that the AWS distributed system is not working as intended, however, there are a few key details to note in these tests. Firstly our AWS instances are all being hosted across the world in us-east-1, this means that the already slow RPC calls from the client are being sent to servers in the USA which introduces a massive amount of latency into the system compared to the locally hosted servers. This could account for the disparity in execution time, but what about the lack of an exponential decay? This also has likely explanations, which relate to the significance of the speed of execution compared to the latency. Whilst we ran the benchmarking multiple times and took an average, there is still massive variation in raw result, probably due to slow downs in network connections. We discuss this issue more in the distributed analysis section, where we look at the overheads for setting up the distributed system and how it reduces as we run more turns.

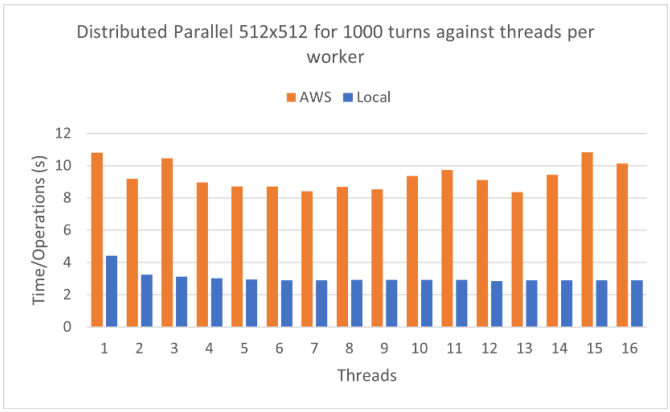

Fig. 5. Graph comparing running distributed system locally and deploying on AWS nodes

D. Handling Multiple Clients

The next stage of development was to allow the broker and workers to handle multiple clients simultaneously and allow them to run their own instance without interference from other clients. This required a rework of how the RPC port assignment was handled, as currently multiple users connecting to the same ports would cause data sharing between server-side channels. To avoid this we used another subscriber model to allow a user to register their existence with the broker. The broker would then create a virtual RPC server on a dynamically assigned port and then return that to the client. This allows the client to redial into their assigned port in order to run their own instance of the game of life. A similar process is also used between the broker and workers, allowing the broker to first check server liveliness, before logging their assigned ports inside the broker's RPC structure.

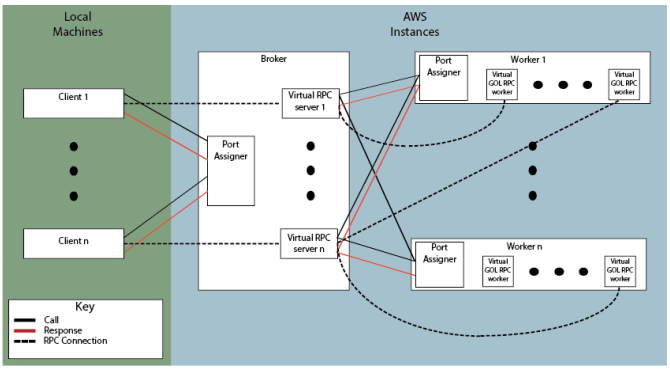

Fig. 6. Diagram of high-level system design

E. Broker Fault Tolerance

An important part of our design is its multiple levels of fault tolerance, these are in place to reduce the number of unhandled client errors and maintain server availability even if partial failure occurs in the network. The first method of fault tolerance on the broker is the use of virtual RPC servers for each client. This encapsulates their interactions, limiting their access to the rest of the server. The main advantage of this is that if a client processes fatal errors, the integrity of other clients servers should be maintained and the server can simply close their connection without completely falling over.

The next instance of fault tolerance is related to how the broker handles workers liveliness. Every time a new RPC server is launched, the broker goes though its list of know worker servers and dials for a port assignment, if these servers don't respond before the imposed timeout then the broker assume that they are down and removes them from the list of possible workers for that client. A similar liveliness check is carried out during the startup of the client's virtual workers in order to ensure that servers are still responding. This process allows the broker to handle partial outages in the worker network dynamically and reduces the chance of errors later in the process, while also allowing for more scalability options. As an imposed limit on this system, if one or fewer workers are active then the system reports this to the client as a network error and terminates execution. The purpose of this is to reduce server load and prioritise existing connections, as if this is ever the case, then most of the worker network is either down or highly unresponsive.

The final layer of fault tolerance in the broker is the error handling and logging, in order to improve maintainability of the system. The broker attempts to catch as many errors before they cause critical failures and instead log them with concise error messages and time-stamping, currently this is output to standard I/O for development purposes, but this could easily be recorded in a log file.

F. Worker Fault Tolerance

Like the broker, the individual workers are also built with fault tolerance in mind. The main focus of the workers is maintaining data integrity upon errors and performing graceful collapses that don't take down other processes running on the same system. The main way this is achieved is through a graceful collapse channel which attempts to halt and send the last stable state before shutting the RPC server in the case of an error occurring.

G. Handling Client Disconnection

A further improvement we decided to experiment with was to add functionality for the user to quit their client and have the workers still run in the background, so that the clients can resume their GoL session after a previous disconnection. This is a form of fault tolerance on the client, as a disconnection will mean the client does not have to make a new connection to the broker and start the workers processing all over again.

We went about this by first disabling the call to the broker that resets the state of all workers when a client quits. The workers will continue to run in the background until their job is complete. We then created a new command-line flag that states a client's id, which is sent along the initial connection to the broker, and used to check against a list of known clients. If this is an existing client, the port relating to the client's connection with the virtual broker is sent back to the client, so that the client can redial the virtual broker that its previous session was being run on. The broker can then use a similar technique to redial the virtual workers for the existing client.

H. Scalability

As with any distributed system, the scalability of its individual components are important to allow it to handle increased network traffic. Our implementation aims to make it simple to add new network nodes without compromising the existing networks functionality. The two main areas fit for scalability are as follows:

H.1. The Broker

As the broker is acting as a middle man between the client and the workers, it is possible to turn it into a network of identical broker instances under the same physical IP. This means that clients are spread across multiple physical machines and if one fails the network doesn't go down completely, it also helps balance the load on individual brokers as clients can be spread across different instances without changing how the system runs.

H.2. The Workers

As our system handles dynamic port allocation, the addition of new worker nodes to the network is as easy as creating them and adding their IPs to the broker's slice of known workers. This, along with the worker liveliness checks, means that new workers can appear and disappear from the worker network without causing broker errors.

I. Distributed Analysis

For our analysis of the different stages of the distributed implementation we had to use different bench marking metrics, as we are no longer focusing on the number of concurrent threads running on each worker. Instead, we decided to measure the average time the network to calculate a single turn of the Game of Life. To do this we ran each instance of the distributed system with 5 different turn length flags and recorded the time taken to complete each job (ns), we then averaged that across the number of turns completed to achieve a metric for Time/Turn.
The four different instances are as follows:

I.1. Dist 1

The single worker implementation of the distributed system.

I.2. Dist 2

The 4 workers + a broker implementation of the distributed system, incorporating halo exchange between individual workers.

I.3. Dist 3 (Local)

The distributed parallel implementation made up of 4 workers each running 8 threads and a single broker. This is being run locally on a single machine [1].

I.4. Dist 3 (AWS)

The distributed parallel implementation made up of 4 workers each running 8 threads and a single broker. This has the workers running on t2.large AWS instances [2] and the broker running on a t2.micro AWS instance [3].

[1] The local machine used for all the tests has 12 logical processors

[2] The t2.large AWS instance has 2 virtual cores

[3] The t2.micro AWS instance has 1 virtual core

The first noticeable pattern in the data is the negative correlation between the time per turn and the number of turns run, this is so significant that Fig. 8. requires a logarithmic vertical scale in order to display average turns times for 10 turns on the same graph as the rest of the data. The conclusion we can draw from this pattern is that the process of setting up the distributed system and sharing the work between the workers has a significant overhead. This overhead is likely caused by the process of establishing RPC connections between servers, especially the startup and ending processes that involve sending large world slices.

This is likely why our distributed system with 4 workers has very similar turn speeds to our 1 worker distributed, as it has to make an additional 3 startup RPC calls, and 4 halo exchange RPC calls each turn compared to the 1 RPC start call of the single worker instance. We can see from Fig. 7. that even after 2000 turns Dist 2 is only marginal faster than our single worker implementation. This, along with the Dist 3 implementation results suggests that even with the startup overheads removed, the Dist 2 implementation is limited by its single threaded workers.

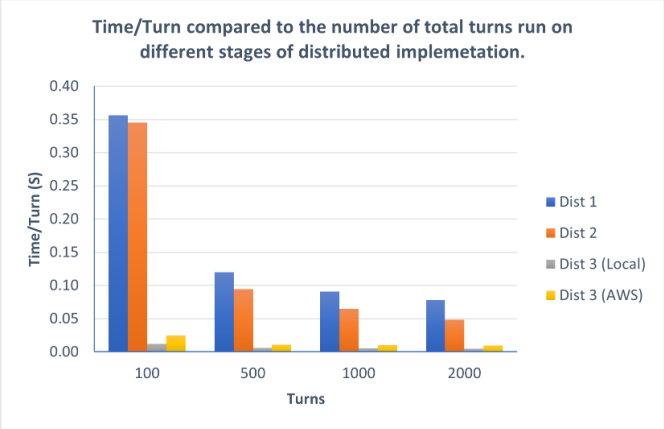

Fig. 7. Graph showing time averages per turn

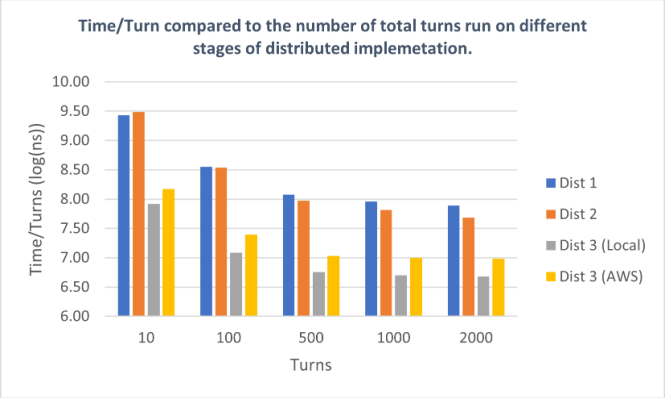

Fig. 8. Graph showing time averages per turn (note logarithmic scale)

J. Possible Improvements

There are a number of possible ways to improve our distributed system beyond simply up scaling the network; below are some we considered, but didn't add due to either time constraints or other limiting factors:

J.1. Checkpointing

Checkpointing is the process of recording intermediary states, which can act as a data backup if a network component fails. We considered adding such functionality to the alive cell checker which occurs every two seconds - the idea would be to also return the current turn's world slice which the broker would assemble with the other workers slices and then save to a variable in the RPC. This would act as fault tolerance if one of the workers went down before finishing as not all of the progress would be lost. This would then allow the broker to either retry from the saved states on the remaining alive workers or return an unfinished request to the user and alert them to this fact.

J.2. Worker Subscribing

This feature would allow workers to dial into the broker once they come online, enabling it to add them to the list of possible servers. This would aid scalability, as new workers can be added during run time; however, it would have the side affect of putting more load on the broker.

4. CONCLUSION

In conclusion, our results have shown us that when comparing our relatively small distributed implementation of the Game of Life to our parallel implementation, the parallel implementations are often considerably faster even for lower thread counts. However, the parallel system has little room for improvement compared to the distributed implementation, which could undergo large amounts of scaling to hopefully improve computation times.