1 Introduction

Thomas Bale (hf23482), Justice Minah (ar23247), Kanghyeon Kim(xx23126). In our project, we were task with the goal of implementing and optimising concurrent and distributed implementations for Conway's game of life. To achieve these results for the concurrent implementation, we made use of deadlock and race condition free code, utilising both channels and shared data. For the distributed implementation, we use AWS nodes accompanied by a broker and RPC calls.

Both implementations provide additional functionality: live counters of the current number of cells alive every two seconds, SDL to visualise the state of the game as it progresses, pausing and quitting functions (p and q keys) and PGM output both at the end of each game or with a 's' key press. For the distributed implementation, 'q' will exit the controller and 'k' will cleanly shut down the whole system.

2 Serial Implementation

2.1 First Iteration

This section will also act as an introduction to the terminology that will be used for the rest of the report. Let the size of the world be defined by $n \times n$ and for all the $0 \leq x, y < n$ coordinates in the world, let $w_t(x, y) \in \{0, 1\}$ denote the state of a cell in the world on turn t at coordinates (x, y). (Where '0' is dead and '1' is alive). For $-1 \leq i, j \leq 1$, we let $n_t(x, y) = \sum w_t(x + i, y + j)$ where $i \neq 0$ and $j \neq 0$ and out of bounds values wrap around. Naively, we first computed:

$w_{t+1}(x, y) =$

$$\begin{cases} 1, & n_t(x, y) = 3 \\ 1, & n_t(x, y) = 2 \ \& \ w_t(x, y) = 1 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

With this, we iterated over each cell, calculating n_t. We can then apply the (1) to obtain $w_(t+1)$ for each cell. This gives a working GoL (game of life) implementation, however it is naive with respect to neighbour calculating [see 2.2]. Further, it reveals that the computation for each cells is identical and independent (if cells are not updated into the original data structure (data race)), thus we can apply the same function at the same time across many cells [see 3.1].

2.2 Serial Optimisations

A note that unless we specify otherwise, the words performance and optimisation always refer to time/speed. An easy optimisation is implemented by analysing how we are calculating neighbours. Consider $w_t(i, j)$ not as a cell, but as a neighbouring cell: it's value will considered by all $w_t(x+i, y+j)$ where $-1 \leq i, j \leq 1$ (not including itself). As such, efficiency can be increased by separating the neighbour calculation. We instead iterate over all cells and if a cell $w_t(x, y) = 1$ then we perform $n_t(x + i, y + j) = n_t(x + i, y + j) + 1$ for $-1 \leq i, j \leq 1$ except $i = 0$ and $j = 0$.

3 Concurrent Section

3.1 First Implementation

3.2 Goroutine Interaction

We can classify the types of goroutines used into two categories: workers and utility. Worker goroutines exchange data with each via a global matrix with a mutex (representing w_t). For neighbours, they all add to a channel which is combined serially in the main thread once execution has finished.

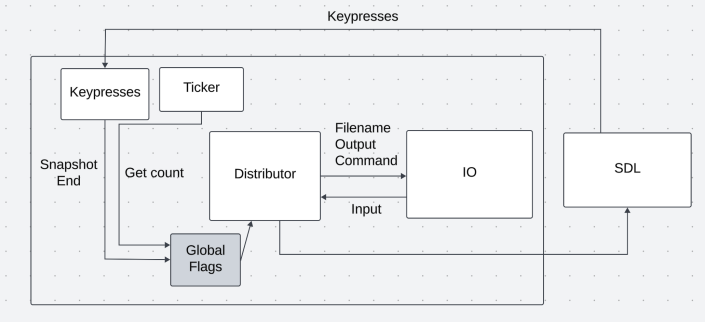

Figure 1: Displays how goroutines interact

The utility channels include: a goroutine to hold a ticker, a goroutine to handle keypresses, and a goroutine to handle IO. The IO goroutine is called by the gol main function on the main thread and is passed command, filename, input, output and idle channels. It runs a continuos loop, waiting for input, output, or idle commands on the command channel. Upon receiving an input command it calls the relevant function which reads from a PGM given on the filename channel. It puts the uint8 image data from the PGM onto the input channel as a 2D slice. Upon receiving an output command on the command channel it writes into a PGM file, given by the filename channel, by taking data the 2D slice off of the output channel and writing byte by byte. The idle command and channel are used to check when input and output are done.

The ticker goroutine is called by the main thread before GoL execution starts. The goroutine flips a global flag every 2 seconds. This flag is checked on the main thread after each turn. When the flag = *true*: $\sum w_t$ is printed for the most recent t and the flag re-flipped.

The keypresses goroutine, similarly, is called by the main thread. It takes the same input, output, filename, command and idle channels as IO. It continuously checks for keypress: if 's' is pressed, the goroutine flips the relevant global flag. The same occurs for 'q', with the addition of the function returning (ending). Both flags are checked on the main thread after w_{t+1} is calculated. When the snapshot ('s') flag = *true* an output command is put on the command channel, filename is given with the size of board, number of turns completed and w_t is passed to the output channel. When the exit ('q') flag = *true*, a snapshot is taken in the same way as above and the program quits gracefully. Refer to the second paragraph to see how IO handles these additions to the respective channels.

The pause functionally, differs from the other two. Upon a 'p' key press: a 'pause' wait group is incremented, causing the 'pause.Wait()' on the main thread to be activated. The

'pause.Wait()' stops execution of w_{t+1}. A new function is also called, serially: running a continuous loop checking for keypresses. 'q' will return from both functions and set the end flag to true. 's' will run the same function described above to give output (however, it runs on the keypress thread instead of the main thread). 'p' will return out of the current function and call 'pause.Done()' to allow w_{t+1} to be calculated and both thread return to normal.

Additionally, the main thread take input to start GoL execution, by: sending an input command to the command channel, sending a filename based on the parameters given and then reading the 2D slice from the input channel to get the starting world. Refer to the second paragraph to see with how IO handles these additions to the respective channels.

3.3 Scalability

This section discusses how program efficiency scales with the number of workers. Please note for the rest of this report we only use two machines for testing. We will denote the machine running (GoOS: Darwin., GoArch: arm64), 8 cores, 8 threads, M1 Pro Mac Chip as 'M1 Mac'. We will denote the machine running (GoOS: Linux, GoArch amd64), 20 cores (8 P-cores + 12 E-cores) and 28 threads, intel i7-14700 as 'Lab Machine'. All test are run on 512x512 board size with 1000 turns.

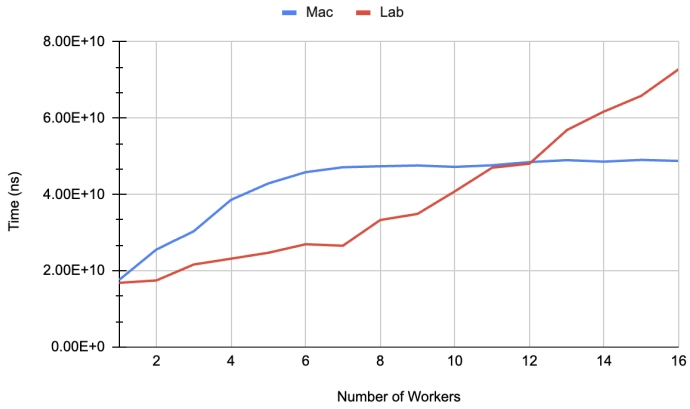

Figure 2: Initial Scaling Benchmark

These initial results are disappointing. We see that the initial implementation does not scale well. Considering Amdahl's law, we can suspect that we have a lot of overhead with initialising workers and in fact workers (parallel work) do not carry a large portion of the workload. See [4.1] for improvements. The above diagram also illustrates the effect of the number of threads on the performance of the system. We see that Mac is forced to start hyper-threading (where one thread acts as multiple) earlier than the lab machine. This causes an earlier plateau in processing time, over the lab machine which has enough threads to not hyper-thread at all.

3.4 Performance Variability

Consider the two different algorithms for edge wrapping shown below and explained next: For if statements we simply check to see if x+i or y+j will be out of bounds and correct them if they are, instead of the two modulus statements. Figure 3 shows clearly, a discrepancy beyond the performance difference mention prior. The two different algorithms do not run differently on the Mac machine, but do on the Lab machine. This is the first sign that code must not be optimised in general but also for the machine and architecture it runs on. For this report, optimisation is for the Mac machine unless specified otherwise.

Algorithm 1 Using Modulus Operator

Input: some $w_t(x, y) = 1$
Output: all appropriate n_t have been updated

```
1:  function EdgeCases(x, y, numWorkers)
2:      for i ← −1 to 1 do
3:          for j ← −1 to 1 do
4:              if i = 0 and j = 0 then
5:                  Continue
6:              end if
7:              X ← (x + i) % numWorkers
8:              Y ← (y + j) % numWorkers
9:              n_t(X, Y) ← +1
10:         end for
11:     end for
12: end function
```

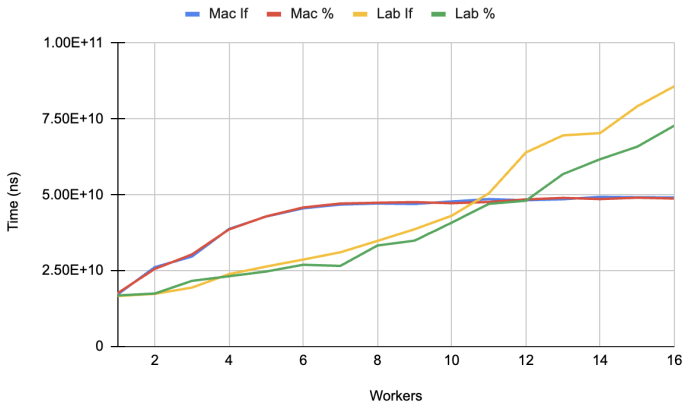

Figure 3: Modulus vs If for arm64 and amd64

4 Concurrent Optimisation

4.1 Worker Communication I

In the first implementation, buffered channels are used for workers to communicate with the main thread. Once all workers have finished calculating their portion of n_{t+1} the data is then recombined serially by the main thread. We can improve this slightly be combining whilst processing. We see in figure 4 that, although it is better for threads ≥ 4 it is worse for low thread counts. This is consistent with what we expect as the combining is likely to be idle when calculating when thread count is low and the channel is not constantly full.

To achieve this whilst workers are still processing, we need to know when workers are finished. Initially, the length of the channel is used, but now channel length dynamic. Since in practice we use uint8s to hold cell data ($255 = 1 = alive, 0 = 0 = dead$), we can use a different value to indicate that a worker is finished. Let this be $finished = 1$.

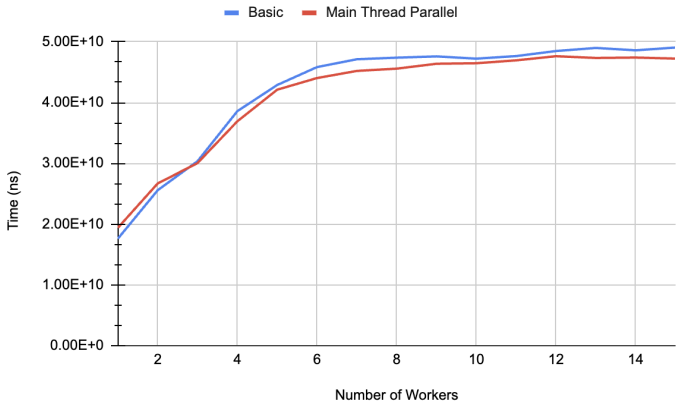

Figure 4: Calculate on main thread

4.2 Non-Worker Overhead I

By non-worker overhead, we mean the computation required to process GoL that is not parallelised. This consists of work: to make or manage workers and serial processing for w_{t+1}. We can reduce some worker related overhead by not destroying and recreating workers inside one t. We now have one set of workers for all processing and wait inside them instead of outside in the main thread. Figure 5 shows a clear decrease in performance [see 4.3] for solution.

Figure 5: Reduced non-worker overhead

4.3 Worker Communication II

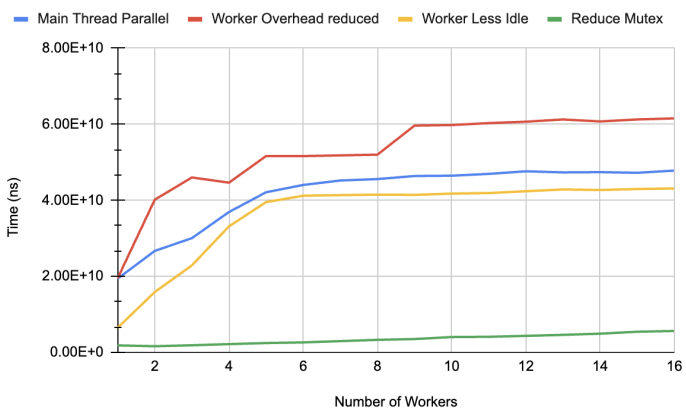

Figure 6: Reduced non-worker overhead

The above solution still involves a lot of waiting and can explain why we see the results (red). Let us instead have workers share no data until we perform $\sum s_{t+1} = w_{t+1}$. We assume that by removing waits, we increase speed . Figure 5 confirms this (yellow). Given reducing the number of waits gave a performance increase, we reduce the number of calls to structs with a mutex where possible too. We do this by passing the world through functions instead of a global world with a lock or with an unbuffered channel. This resulted in a large increase in performance (green).

4.4 Communications Overhead I

When talking about communications overhead, we are talking about both the number and size of data passed between methods. To optimise in this area, we should think about both: what we are sending, and how we are sending it.

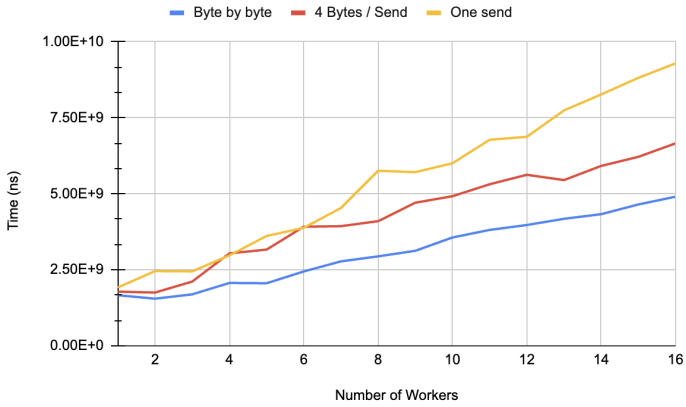

Figure 7: Workers sending varying packet sizes

Now, we consider how we are sending information. In our initial approach, workers send data (uint8s) as soon as it is processed (byte by byte). In our most recent approach we are returning each workers results as one packet on a channel. We would like compare performance for these and intermediate packet sizes. To achieve this, each worker will combine uint8s into larger formats and send those over the channel. The data is then unpacked serially and updates are made to the board. Figure 7 shows that the packing process does not outweigh the cost of communication here as it does not scale well.

4.5 Communications Overhead II

When considering what we should send, it is obvious that most of the inter-method communication is workers sending via channels. To optimise, workers should not send data if a cell does not need changing. Our initial change was to the rules by which our cells are updated:

$$w_{t+1}(x,y) = \begin{cases} 0, & n_t(x,y) > 3 \text{ or } (n_t(x,y) < 2 \\ 1, & n_t(x,y) = 3 \\ w_t(x,y), & \text{otherwise} \end{cases} \quad (2)$$

The above adapted rules do not completely eliminate unnecessary sends, namely when: $(w_t(x,y) = 1$ and $n_t(x,y) = 3)$ or when $((n_t(x,y) > 3$ or $(n_t(x,y) < 2)$ and $w_t(x,y) = 0)$. However, it is important to note we are not computing more checks than the previous rules used.

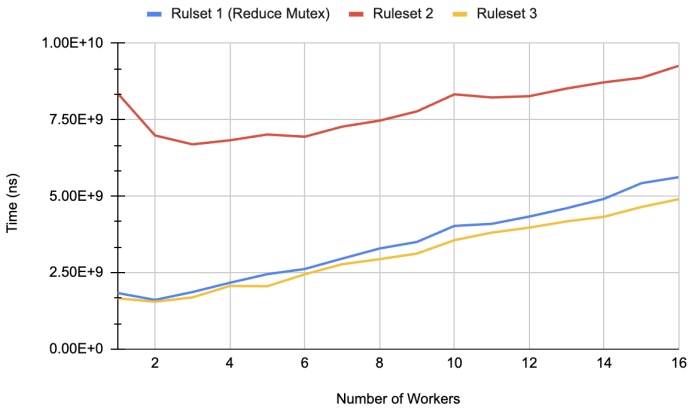

Figure 8: Performance with different rule sets

By changing the rules to achieve a rule set that only sends data when a change occurs, we produce:

$$w_{t+1}(x,y) = \begin{cases} 0, & w_t(x,y) = 1 \\ & \& \ (n_t(x,y) > 3 \text{ or } (n_t(x,y) < 2) \\ 1, & w_t(x,y) = 0 \ \& \ n_t(x,y) = 3 \\ w_t(x,y), & \text{otherwise} \end{cases} \quad (3)$$

With the above, we make 5 checks instead of 3 but, we save largely in communication overhead. As spoken about before in previous sections, we have been able to optimise the first rule set without the use of channels. The second two, new rule sets benefit from channel use over writes with a mutex, however, due to less sends (also covered above).

4.6 Scalability II

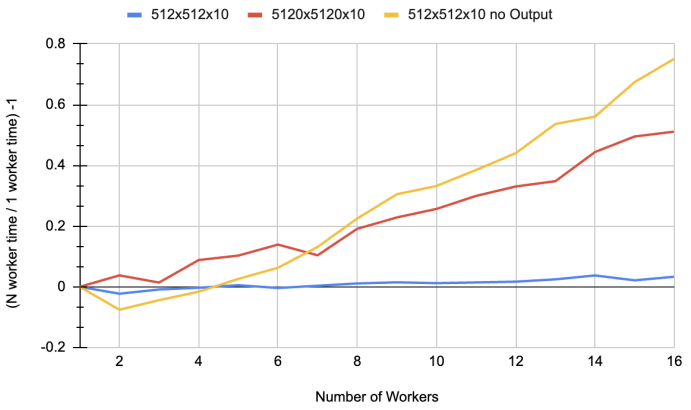

Figure 9: Considering amount of parallelism

See now, that even when all GoL processes that can be parallelised are parallelised [see fig 8 yellow, fig 6 green], we still do not scale well with the number of workers. We suspect one or both of the following is occurring: calculating w_{t+1} does not account for a large portion of overall processing (poor parallel performance in accordance with Ahmdal's law) or the Mac machine choses to hyper-thread before using all available threads (concurrent processing is not taking place but we increase overhead with the number of workers regardless). We test the former by processing with a larger image size : 5120 x 5120 for 10 turns and compare with 512 x 512. We find It scales more poorly, perhaps this is due to larger IO overhead - removing output shows worse scaling also, so this cannot be the case [fig 9].

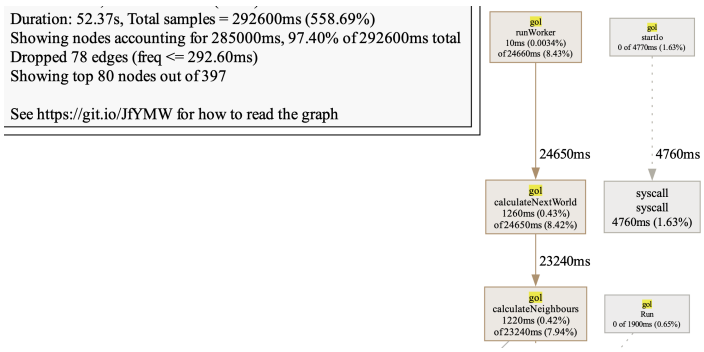

Figure 10: Portion of work for all workers

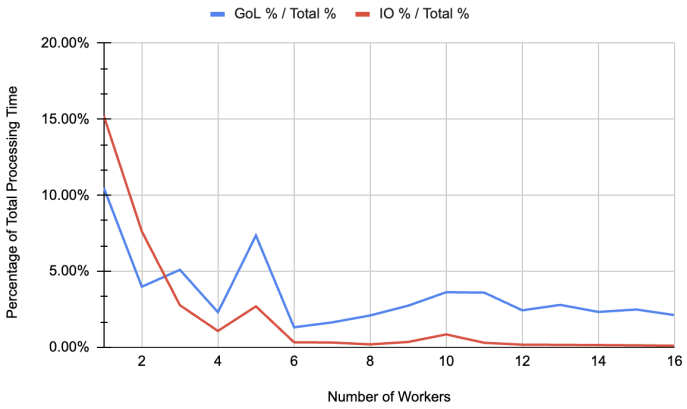

Figure 11: Comparing worker portions

Figs 10,11 make use of pprof and give reason for our results and disproves what seems to be a scaling issue. We see that both the GoL computation and IO reduce as percentage of total processing time with added workers. Despite repeat tests, the spikes at 5 and 10 persist. Given that IO % also increases, we assume this is not a by-product of the code but rather there is a change occurring with how the Mac cores are being used.

5 Distributed Section

5.1 System Design

For the distributed section, we make use of a broker/server and separate nodes. Each node contains two types of functions: functions to calculate w_{t+1} from w_t and functions for communicating with the broker. The broker, has 4 types of function: communicating with workers, running GoL, communicating with controller, GoL utility. The controller has four: IO, communicating with broker, utility, initiating GoL.

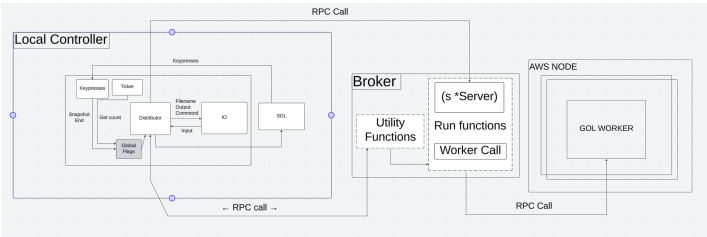

Figure 12: Distributed interactions

We run as following: workers are initiated on ec2 instances and listen on a specified port. The broker is given the list of ip address and ports for the workers. The broker is initiated and listens on a specified port. The controller is given the address and port to dial for the broker and dials upon initiation. The controller RPC calls the broker to start GoL and receives a response with finishing details for t = total turns. The broker will dial all workers and execute all turns of GoL with workers being given the relevant s_{t+1} to calculate from w_t.

5.2 AWS vs Local

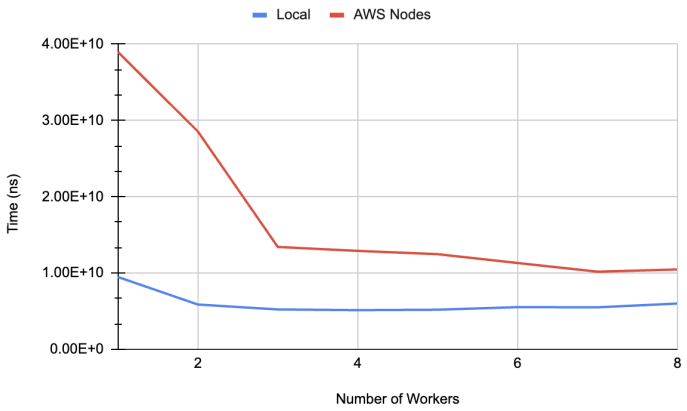

Figure 13: Local vs AWS performance/scalability

We test the scalability and raw performance of both running on AWS ec2 instances and running the system locally. For AWS, we run the server and broker on one instance and have one instance per worker. Figure 11 shows good scalability for both environments. As suspected, it runs much faster locally.

The data implies that in the AWS environment, the point at which more threads becomes less beneficial is much earlier than locally. We can speculate that this is due to a difference in the distribution of performance costs. The main performance cost, locally is overhead relating to creating and managing workers and their data. For the AWS environment the main overhead is data being sent between the distributed parts.

5.3 SDL Live View

Here, we compare two implementations against two non-SDL implementations. The first is the basic implementation described in [5.1]. To send flipped cells, we change the rules that process s_t to s_{t+1} (this is not an optimisation [see fig 12]). We first check the state of s_t, then run GoL Logic:

$s_t(x, y) = 1$ then $s_{t+1}(x, y) =$

$$\begin{cases} 0, & n_t(x, y) > 3 \text{ or } n_t(x, y) < 2 \\ 1, & \text{otherwise} \end{cases} \quad (4)$$

$s_t(x, y) = 0$ then $s_{t+1}(x, y) =$

$$\begin{cases} 1, & n_t(x, y) = 3 \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

This way we can insert a flip cell in both non 'otherwise' cases. We can can then send this back in a response to the server. The server can in turn RPC call to the distributor to send the cell flipped command. (This requires the distributor to listen and server to dial upon initiation). We also implement the live view by running a cell-wise xor between the w_{t-1} and w_t in the broker and RPC call the same distributor function. We save on communication overhead here but increase work inside the broker. The results [fig 12] show that the former is marginally faster, however, we speculate and expect if we benchmarked for larger numbers of workers, the later would perform faster.

Figure 14: Comparing live view implementations

5.4 System Faults

Disconnection of a worker outside of processing is spoken about in [5.5]. Disconnection of controller is spoken about in [5.5]. Disconnection of broker will result in failure of the system. We could mitigate this issue with redundancy and failover [6.4].

5.5 Fault Tolerance

If a local controller is disconnected, a flag, w_t and t are held in a struct in the broker. A flag can be passed to newly created local controllers to ask to continue from the stored w_t and t. This only happens if both flags are true.

Disconnection of a worker between calculations: If a worker has been disconnected between turns, it does not impact the system as workers are dialled for each turn. This was implemented but we do not run it due to performance considerations [see 7.3].

6 Distributed Optimisation

6.1 Re-dialling Workers

This is part of the fault tolerance [see 5.5] and we would like to see the performance impact this has on our system. Comparing locally, we see that for high worker numbers, the raw cost does not seem large [fig 13].

When considering this as a percentage increase [fig 14], however, we find that this cost is significant. Cost starts at 40% and by 5 workers, a 100% increase in cost is reached. Throughout all testing, we had no errors from disconnected workers locally or on AWS. As such, we conclude that the

performance costs associated are too high, given the exceedingly rare chance of error (ostrich strategy). Peculiarly, we see the opposite for AWS vs locally in both raw and percentage change data. Namely, that redialling was faster than dialling once. We have no concrete explanation for this result, but speculate that this may due to faster routes for data packets to be sent across being discovered with redialling.

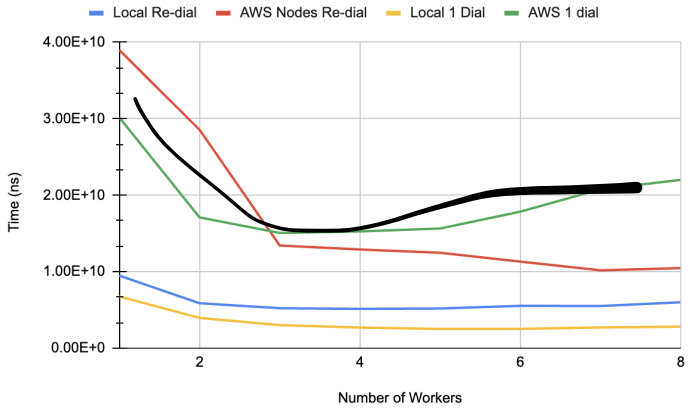

Figure 15: Cost of redialling workers

Figure 16: Percentage increase of cost

6.2 Parallel Distributed System

Here we implement a parallel system on each distributed node. We use the fastest running code from the concurrent section [see fig 8, yellow] and adapt slightly to split up a segments of the world into smaller segments. We test locally for different permutations of workers and threads. By looking across top left to bottom right diagonals, we find that for any one, an equal number of workers and thread is optimal [Fig 17]. The most optimal being 4 x 4. We do note that the graph is slightly weighted towards more workers over threads being better. This can be explained with the same reasoning in [4.6].

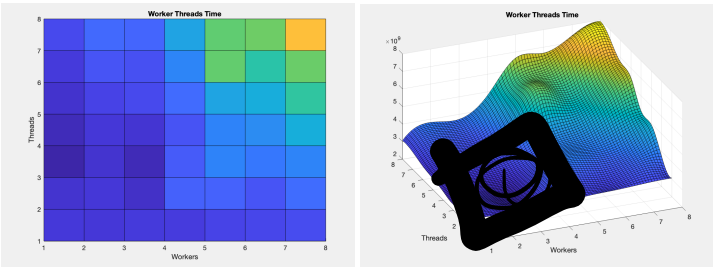

Figure 17: Different permutations performance

First note that the results for AWS are more sparse due to time restraints on benchmarking. We see again that the data shows better performance around the workers*threads = 16 mark. We note, however, that unlike locally, a real distributed system does not favour more workers, with no amount of threads in 4 workers out performing 2 workers. Similarly, have poor performance on both ends of the thread spectrum, unlike locally, where we only have poor performance with high thread counts. Both are to be expected as the communication overhead in our distributed system far outweighs other performance factors.

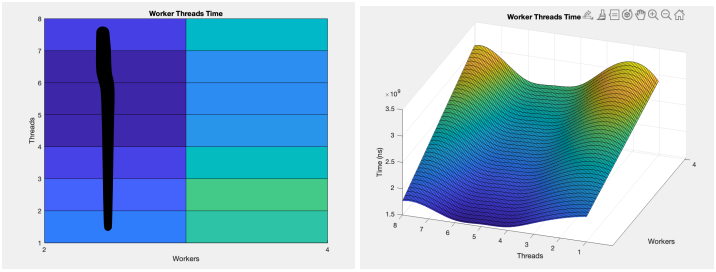

Figure 18: Different permutations performance

6.3 Halo Exchange

We implement halo exchange on our parallel distributed system with live view. The broker already holds information on the node's addresses and ports and take advantage of this by connecting nodes inside the broker. Each node $n \in \{2...N-1\}$ dials the $n+1$ node and listens to accept the $n-1$ node. The Nth node dials the 1st node. Through RPC calls, requests and responses, the additional neighbour counts are added to the start and end of each worker. Halo workers inter-node for any number of threads one 1 node.

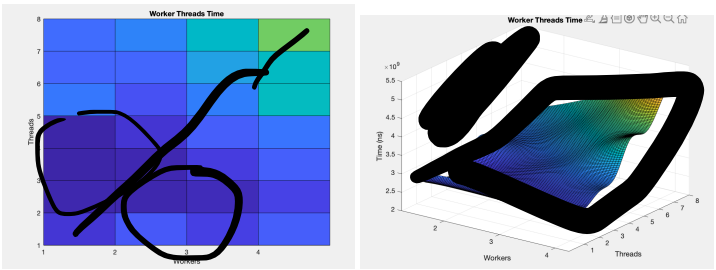

Figure 19: Different permutations performance

7 Other

7.1 Obtaining Results

All graphs in this document are produced on data ran by go benchmark tests. The benchmarks are run three times per data point and the average is used to plot graphs in excel and matlab.

7.2 Further Improvements

This section covers improvements that were considered but not implemented. We can implement redundancy and failover wrt the broker. By running a spare redundant broker, if the main broker fails we can switch without killing the system. We can ensure nodes (including controller and broker) are in different geographical locations, this way if one node fails, it is less likely others will follow.