Conway's Game of Life in Concurrent and Distributed Systems

James Millan (nf20792), William Tripp (iq20064)

1 Concurrent Implementation - Functionality & Design

1.1 Serial Implementation

As a starting point for the project, we created a serial implementation of Game of Life (GoL) that was not parallelised. It read in the image, represented as a 2-D slice from the IO go routine by first sending the IOInput command down the Io channel followed by the filename. Next, it iterated through all the turns requested, updating the slice on each turn. Finally it returned a Final Turns Complete event down the events channel which is part of the SDL go routine. This event struct consisted of a slice containing the coordinates of all the alive cells in the final iteration and the number of turns completed.

1.2 Concurrent Implementation

After creating the single thread solution, we moved onto designing a parallel solution that splits up the work for each turn amongst multiple worker go routines (the number of which is supplied as a threads parameter). The important problem to solve was ensuring each worker had access to the correct number of columns. A naïve implementation is to divide the width of the world by the number of worker threads. However, in most cases, these two numbers will not divide evenly. Golang's remainder operator was used to ensure the division worked without the number of threads being a power of two. For remaining columns x, the first x workers are given an additional column. This is more efficient than a previous version in which all the remainder columns where supplied to one worker. This doesn't maximally optimize spreading the work across all the worker go routines. Next, all the workers send their updated slice back to the distributor. For each turn, these were appended into a big 2-D slice (nextWorld). After that, the main loop updates the currentWorld values for each cell and sends a Cell Flipped event to the events channel. Finally, after each turn is completed, the distributor has to send a turn complete event down the events channel.
The next step was to implement a ticker that sends an Alive Cells Count event down the events channel every two seconds. The most basic way to implement this was to create a ticker that ticks every two seconds, update the main function into a select statement and include a case for if the ticker ticks. Otherwise, the default case should be the logic described in the paragraph above.
Additionally, it was necessary to implement key presses for control rules. Again, update the case statement in the for loop to include the case where a key is pressed. The key press is passed to distributor by the SDL go routine via the keyPresses channel. This allows for saving an image at any point, pausing the program, and safely writing the image and quitting. All of these commands work whilst the system is in a paused state.
Once all the turns are completed, the board must be converted to a pgm file. This is done by the Write File function. This sends an IO Output command down the IOcommand channel followed by the filename down the IOfilename channel. It then sends the world byte by byte down the IOoutput channel. It checks if the IO go routine has finished writing the image using the IOidle command before sending an Image Output Complete event down the events channel. Finally, the system checks once again that all IO has finished, then sends a State Change event down the events channel that says the program is quitting before closing the events channel. This solution is free from deadlocks and race conditions. Note that although this solution doesn't use any mutex locks, it is not a pure channels solution. This is because we pass a pointer to the memory containing the board down channels rather than the contents of the board byte by byte down the channels.

1.3 Concurrent Implementation (Mutexes and additional go routines)

The methods described above to implement the ticker and the key presses were crude and did not technically trigger instantaneously. If a board was arbitrarily large, then the ticker could be ignored for a number of seconds, then the Alive Cells event will not be triggered every two seconds. This could happen because processing a turn could take more than two seconds, since the distributor is stuck waiting for this turn to be complete, one can question the correctness of the program. A better solution uses mutex locks to lock two key data elements; the current world and the turn counter. This allows the ticker function and key presses function to be run as go routines. These functions

send the events directly down the events channel rather than doing so in the main for loop of distributor. This solution is free from deadlocks and race conditions.

1.4 Concurrent Implementation (Halo Exchange)

A halo exchange system substantially decreases the communication overhead of ensuring correctness of the workers by having the workers communicate their halo regions to each other directly rather than reassembling the world every turn and redistributing it. This was implemented by creating channels between workers passed as structs to each worker's go routine. As workers must both send an receive halo regions, a system is required to stop the system from deadlocking if both a worker and its neighbour are both attempting to send or receive at the same time. To avoid this, each worker is assigned to either send or receive their halo regions first, this Boolean value will be the opposite of their neighbour's. As the workers are only dependent on their direct neighbours to progress, workers may be processing different turns at different times, which is an issue when events such as GetAliveCells require a consistent state. In order to achieve this, each turn the workers are blocked by a channel, which is unblocked by a loop in a central thread in order to synchronise them and then make event calls.

2 Concurrent Implementation - Problems Solved

The serial implementation solved the initial problem of reading in the image from the IO channels. Similarly, it iterated through all turns of GoL before returning the coordinates of the alive cells on the final turn.
The parallel implementation then made each turn more efficient by splitting the work for each turn amongst multiple worker go routines. The function slice world takes care of giving each worker the 'halo regions' necessary to calculate the next state of the edge cells on the board and at the edge of the slice given to it. This is fundamental to the logic of GoL since the board is a closed domain, so the edges on opposite sides of the board need to be connected. The bound number function is necessary when calculating the number of alive cells surrounding a current cell because Golang's '%' acts as a remainder operator rather than a modulus operator therefore it can return a negative number. Furthermore, this system solved the problem of receiving the number of alive cells every two seconds using a ticker. Similarly, the implementation of key press control rules was solved by interacting with the SDL go routine via the keyPresses channel and using a select statement on that input. In addition, the distributor interacted with the SDL go routine via the events channel to send all the events that enable visualisation of the program and testing to ensure correctness of the program. Finally, it interacted again with the IO go routine to write the output of the program (i.e the final state of the GoL board) to a pgm file.

3 Concurrent Implementation - Critical Analysis

3.1 Benchmarking

We benchmarked the 512x512 pgm file for 1000 turns. The benchmark was run 10 times to try and eliminate Gaussian noise from other programs running on the computer. We used the linux lab machines on Intel i7-8700 processors containing 12 logical cores which is an increase from the personal machines. All benchmarks were conducted on the same day, on the same machines, using the same commands. In other words the conditions were as controlled as possible and the only variable changing was the program being benchmarked.

3.2 Serial Implementation vs Initial Implementation

Clearly demonstrated in Figure 1, when: $n(workerthreads) > 1$, the parallel implementation reigns supreme over the serial implementation. This is due to splitting up the work amongst multiple go routines that work concurrently to process a smaller slice of the board, rather than processing the entire board sequentially like in the serial implementation.
In contrast, the serial implementation is faster than the parallel for a single worker thread. This is because in the serial implementation you do not have to:

- Calculate the slice of the board to distribute to each worker.
- Retrieve each slice from each worker and store that.
- Send an Alive Cells Count event every two seconds.

Although there is only one worker, the additional overhead from these three operations makes the parallel implementation slower. Whilst there are other operations the parallel has added over the serial, these will not have a significant overhead that would majorly contribute to the result found.

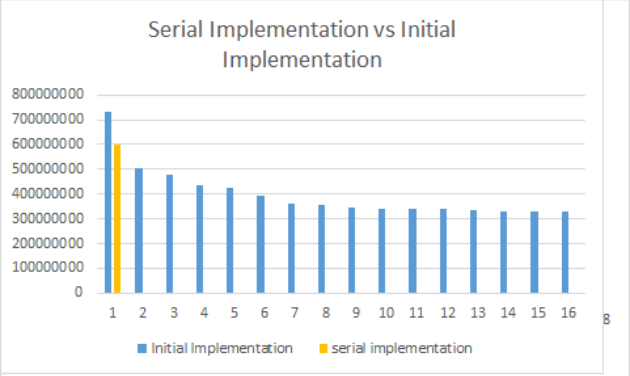

Figure 1: Benchmark time for the serial implementation and the first parallel implementation for 1-16 worker threads

3.3 Initial Implementation vs Mutexes Implementation

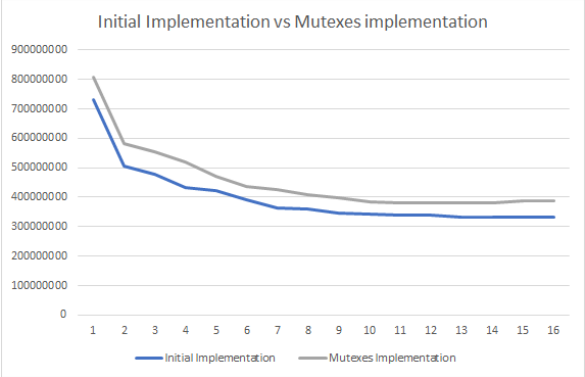

Figure 2: Benchmark time for the inital implementation and the mutexes implementation for 1-16 worker threads

In Figure 2, the results strongly indicate that the original parallel version is a more efficient implementation. Despite the fact it's main for loop runs as a select statement so the program must wait for the ticker and key presses to complete before continuing. The overhead of the two mutex locks used in the mutex implementation. Since locking and unlocking a mutex lock is cheap in terms of performance, one can deduce that the distributor must be waiting for access to one of the mutex locks regularly. The world mutex lock needs to be accessed by the main for loop on every iteration when it updates the current world. Likewise, the turn mutex needs to be accessed every turn to update the turn counter. The ticker needs both of these mutex locks every two seconds and the keyPresses needs them when writing a file. Undoubtedly, there will be times when the for loop must wait.
Nonetheless, The shape of both graphs are very similar, displaying that the runtime decreases due to the additional workers in both programs. The graphs show that runtime has an inversely proportional relationship with the number of workers, due to the processing load of the game's logic being split between workers which execute it concurrently. However, beyond a certain point, adding additional workers yields little to no improvement to performance. This is because the program was benchmarked on a 12-core processor, meaning that if there are more than 12 workers, they cannot each be assigned a unique thread to execute on, which means that not all processes execute in parallel. Additionally, with each increase in workers, the distributor has to do additional work:

- Calculate the slice of the board to distribute to each worker.
- Retrieve each slice from each worker and store that.

This additional overhead causes the increased performance of extra workers to diminish. For a sufficiently large enough number of threads, $16 < n \leq imageWidth$ the performance will not be increased but will begin to decrease. This is because the board has to be pieced back together from a large number of slices, and the slices are so small they are computed very quickly and then the largest performance impact comes from piecing it back together.

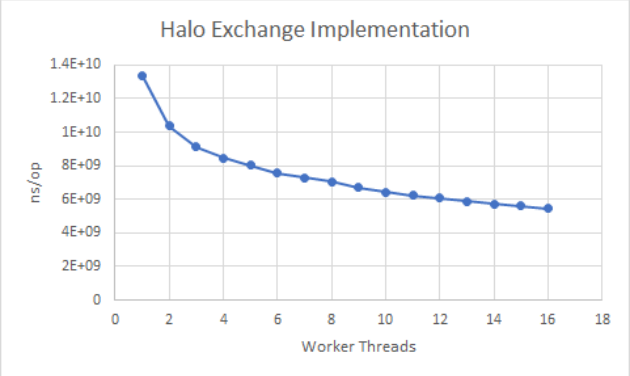

Figure 3: Benchmark time for Halo Exchange 1-16 worker threads

3.4 Halo Exchange Analysis

As figure 3 highlights, the Halo Exchange implementation parallelises greatly to obtain a graph exactly as expected. However, due to issues with implementation it runs a factor of 10 slower than the other parallel implementations. We speculate that this is because the workers are blocked by a channel that is only unblocked by the main loop when every worker has finished their turn. This had to be implemented to make the Test Alive tests pass so that the workers were synchronised. Another issue is that sending the halos between workers is a blocking operation that can increase the performance overhead of the system.

3.5 Potential Improvements

In order to improve the halo exchange version, we would ideally like to remove the blocking calls between the main loop and each worker. This could be done by using buffered channels so the workers can go at any speed and just send their halos and access the halos whenever they want. To conclude the Parallel section of this report, it is clear that the parallel implementation made an improvement in terms of performance and features over the serial implementation.
The initial parallel version was more efficient than the mutex version because of the fact that the distributor has to wait repeatedly for the mutex locks. The halo exchange version scaled the best but was quite inefficient due to the implementation details of the system.

4 Distributed Implementation - Functionality & Design

4.1 Basic Controller Implementation

The idea behind the distributed system was to have the GoL work split up across multiple workers on different machines using RPC calls. When creating a more advanced system, the need for a broker that interfaces between the controller and the workers becomes much more desirable. Consequently, we opted to begin with using a broker, rather than a more simple server-worker model. The broker uses a publish-subscribe model, listening for initial subscription request from the workers on a port specified by a flag. The workers then act as clients to the broker, attempting to connect to it on an IP address given by a flag and providing their IP address and open port (also provided via flags) so that the broker can keep track of them in a global slice and make RPC calls to them. Once the workers have confirmed a connection to the broker, they will begin acting as servers that can be called by the broker. This system allows greater decoupling of the broker and workers, allowing additional workers to be added at any point and avoiding the worker's addresses needing to be specified to the broker before execution. The controller (distributor) then reads in the image from IO as described in 1.1. It then dials the broker on a port specified in a config file using a Broker Request RPC call. It supplies the broker with the current world and number of turns in the request struct. It will receive the final world state after the requested turns as a response.
The broker request function works similarly to the distributor described in 1.2. It reads in the board and number of turns and requests and makes the channels for all the workers. On each turn, it iterates through each worker and runs the call worker function as a go routine. The worker function that processes this slice is almost identical to the serial threaded implementation described in 1.1. The updated board is then returned in the response and each slice is pieced together and the program continues onto the next turn.
Once the broker has completed all the turns, it responds to the controller with the final board. The controller then

sends a Final Turn Complete event before quitting. In the code, we deferred the closing of all listeners therefore we do not have to handle closing them; Golang handles this for us.

4.2 Advanced Feature Broker Implementation

The advanced broker implementation includes all of the functionality described in 1.2. The broker starts by initializing the necessary mutex locks and channels for the system to operate. The controller starts Events Routine, a go routine which runs in an infinite loop. It checks for the ticker ticks and keypresses down the relevant channels. It only terminates when a keypress kills the program or all the turns have been completed. The controller then makes a Broker Request to the broker. This function is where the turns of GoL are processed.
At the start of each turn, it calculates the number of alive cells and the turn number so that these values can be given to the ticker every two seconds. The alive cells are calculated every turn, as opposed to only when requested as in the parallel implementation. It was implemented this way due to network delays between the program being registered as finished on the controller and broker, meaning that the ticker in the controller may be able to request alive cells from the broker while no instance of Game of Life is being processed. Therefore, the broker needs to be able to return valid data for a TurnComplete event even after Broker Request is finished, so retrieves the turn and alive cell global variables rather than requesting from the Broker Request directly via channels. A mutex lock is required for these two global variables.
The broker then distributes the workers by calling each one as a go routine, and making sure it gets the correct slice of the world. This is exactly the same as in the parallel implementation. The broker calls each worker as described earlier. The broker then reconstructs the board from all the workers. The broker must then check if the PGM output has been requested (if an s or k is pressed). It requires a mutex to access this. If the condition variable is true, send the board and the turn number down channels and reset the condition variable to false. There is a very similar setup for checking if k has been pressed to, in order to shutdown the system and if p has been pressed to pause the system. The broker then sets the final board and final alive cells in the response and returns.
Finally, the distributor writes the image output and sends the relevant events to down the events channel before closing.

4.3 Fault Tolerance

The implementation described in 4.2 solves the problem across a distributed system. However, if a worker node crashes, so does the entire system. This would not be very effective in a distributed system with thousands of workers working constantly, in a system that must be running all the time but where crashes are inevitable. Therefore, fault tolerance is desirable and we implemented it in the broker.
When the broker request function distributes to the workers, the workers send true down their own safety channels. After this the broker then checks for faults by iterating through all the safety channels and checking for any that are not true. If a fault is found, the broker closes all connections with all workers. It attempts to reopen every connection and only stores the ones that are opened successfully. After this, it distributes work to the workers again, checking for faults again afterwards and then continuing with the for loop.

5 Distributed Implementation - Problems Solved

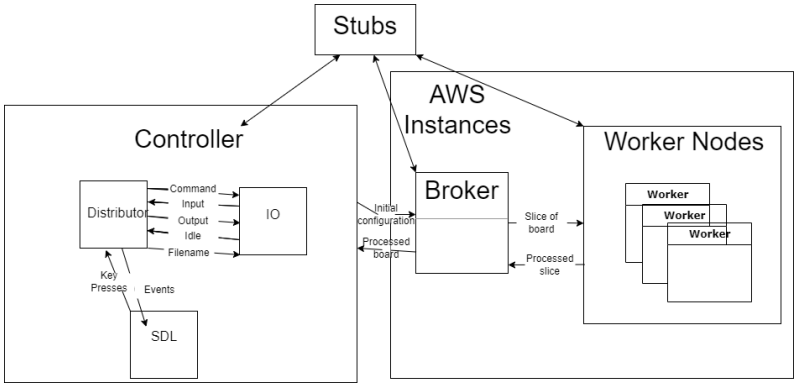

Figure 4: Diagram demonstrating the coupling of the distributed systems components.

This distributed system solves the problem of GoL by delegating the computation to worker nodes across a network with the aid of a broker, using RCP calls. The logic is very similar to that of the parallel implementation. However, the important distinction between the two of them is that in the distributed system, the controller can be run on a computer that isn't very powerful and only marginally affect performance. The majority of the work will be done across the network (in our case on AWS instances) this is very useful if you are interested in using IaaS for your computing power.

6 Distributed Implementation - Critical Analysis

6.1 Benchmarking the distributed system

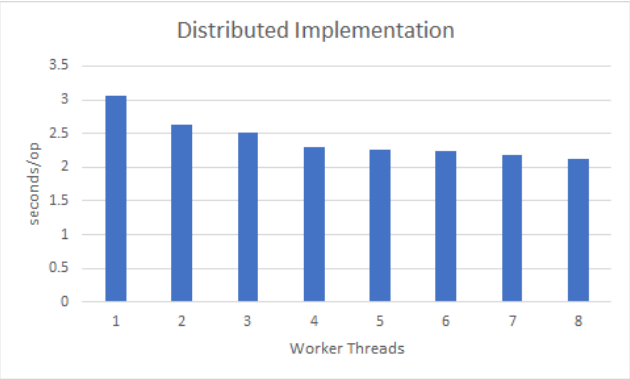

Figure 5: Benchmark time for the distributed implementation for 1-8 worker threads at 512x512 for 100 turns.

The distributed system was benchmarked using AWS t2.micro instances for the workers and the broker. The controller was run on a personal laptop which didn't have a major impact on the results since the vast majority of computation is conducted in the workers and broker.
Figure 5 clearly shows a decrease in runtime as the number of worker threads increases. Therefore this system scales well. This performance improvement for increased numbers is workers is not as good as for the parallel implementation. This is for two reasons:

1. The benchmarking was only run for 100 turns rather than 1000 turns so there was less chance for the performance improvement to be measured.

2. There is a relatively large overhead for using RPC calls in a distributed system like this.

As a direct consequence of (2), with an increase in worker threads, the performance increase will diminish and will eventually become a performance decrease.

6.2 Potential Improvements

From a functionality perspective, the fault tolerance works for all number of workers, except one worker. When this happens, the broker is left with no workers in it's list and it cannot distribute work to no workers. Consequently, it causes the system to fail. Ideally, you would want to receive a statement to the console detailing the situation and the system would go idle whilst waiting for new workers.
In addition, there are a number of global variables and as a result several mutex locks are required. This is not ideal for adding new features to a system for instance, additional control rules.
From a performance perspective, minimize RPC calls. In our current system, the main RPC calls are; the controller sends the broker the inital board, the broker sends the workers their slice, the workers send back their processed slice to the broker and the broker sends the board back to the controller. Finally, the controller also calls the broker when a key is pressed and when the ticker ticks.