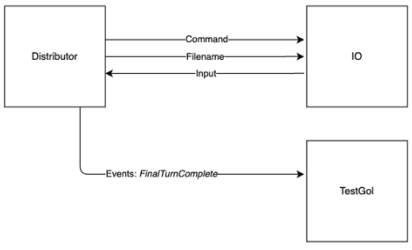
Michael Wyrley-Birch an20504
Emil Sebastian Jino hk20234

# Introduction

In this project we were required to build upon our initial Game of Life program which is simply a 2-valued 2D matrix, where cells can either be alive or dead and returned a pgm image with the alive cells after a certain number of turns. We took this simple code, and we made some Serial, Parallel and Distributed implementations which allowed the runtime of the code to be decreased vastly due to multiple worker threads being used, we shall explain this further in the respective section.

## Serial

We started off by creating a single threaded implementation of the game of life which already had 3 go routines in the skeleton code, that interacted with each other to produce an output, however we were required to initialise the channels to allow channel communication as we could not call methods from the IO goroutine. To allow the serial test to be run we had to send FinalTurnComplete down the Events channel. This diagram shows how the channels are linked.




We tested two implementations for serial one using modulo and one without. We found the modulo operator to have significant overheard, especially for larger images, therefore we decided for all future implementations of the code to calculate the next state, use non modulo.

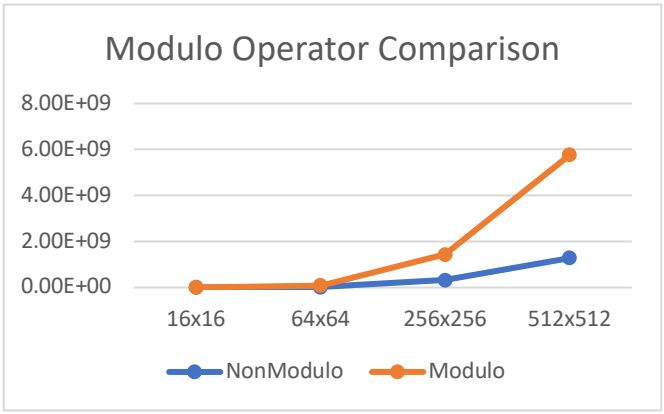
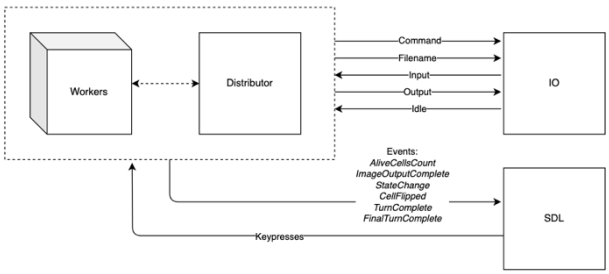



*Figure 1: Modulo comparison 10 trials*

To test the modulus we ran 4 different size images from 16x16 to 512x512 for 100 turns on the lab machine.

## Parallel

Once the serial code was completed, we moved onto the parallel part of this coursework. We parallelised GoL by adding multiple worker threads (number requested in gol.Params.Threads) to distribute the work evenly and thus increase the efficiency of the code and decrease the total run time. Each worker is responsible for concurrently processing a different part of the image.




Our worker go routines (calculateSlice), it takes in part of the world in order to increment the state once. While there is a newWorld we loop through the cells changing the state for that slice. The input world has an extra row on the top and bottom in order to allow the go routine to

completely cover that section of the board. For each item in the world, we check its neighbours (checkNeighbours) which loops through the 8 cells surrounding it to calculate its next state according to the Game of Life logic.

In the parallel section we had 2 go routines other than the workers, which handled the key press inputs (keypress) and the number of alive cells(reportAlive)

Following the steps we next created a ticker which reports the number of alive cells in the world every 2 seconds. We did this in the reportAlive function by sending the gameboard struct which contained the world and turns, down a channel which is in a mutex so that we don't have a race condition with the number of turns and the board state , we then called the getAlive function to get the number of alive cells and sent it to the events to be displayed to the user.

We also implemented user interaction by adding key presses, this allowed the state of the board to be paused, saved as a pgm file or to terminate the program. For pausing the world , in a mutex it will take the current state of the board and notify via the events channel that the state is paused. It will then continue to look for a P key press after which it will notify once again via the events channel, that there is a state change.

For terminating the program, similarly to pausing it outputs the current state of the board and then notify via the events channel that it is terminating.

The keypress S just saves the board as a pgm file.

We then worked on outputting the state of the board as a pgm image, we sent an iOoutput command down ioCommand and filename down ioFilename, then sent the world byte by byte to the ioOutput this then produced a pgm image of the board state.

The final stage our Parallel implementation was to send the CellFlipped and TurnComplete events to the user. Our CellFlipped event was called in the calculateSlice go routine (worker), this tells the GUI about the change of state of a single cell. The TurnComplete was sent in the calculateNextState notifying the GUI of the completion of one turn.

We decided to test two implementations of the code, one where we sliced the world into the correct size for each worker and one with the world sent as a whole. On a lab machine we ran both methods on a 512x512 image for 1000 turns. This allowed us to get an a better idea of the difference in speed.

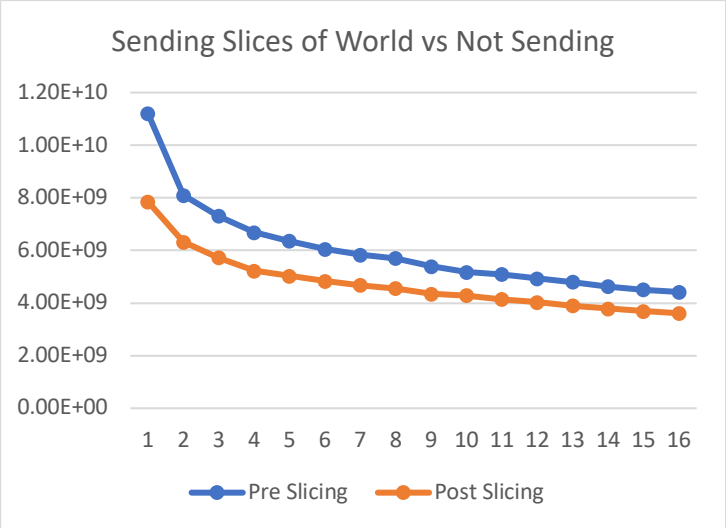



*Figure 2: World sent as slice vs sent as whole 10 trials*

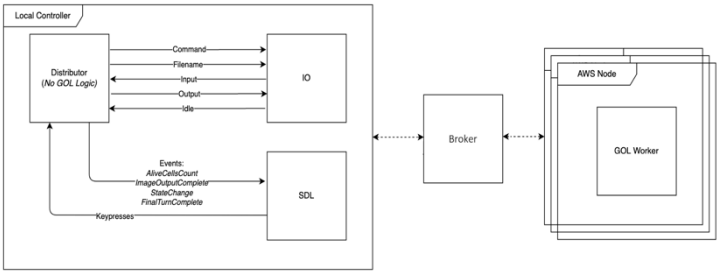
From our benchmarking, the results we obtained are shown in Figure 2 and we can see that sending the correct size slice to the workers yielded a significantly

faster runtime. This is due to the overhead that is present when sending such a large slice down the channels which far outweighs splitting the slice into smaller chunks and then sending it.

The curve in Figure 2 is produced as the number of workers increased, the runtime decreased. With 16 worker threads, the post slice filter was 2.2x faster than the serial implementation with 1 worker. The perfect improvement would be a runtime that halves as the number of threads is doubled. This is not the case here, because we measured the total time taken to apply the filter and output an image. The workers apply the filter in parallel, but the image input and output, which happens outside the workers, is not parallelisable. The reason it is exponential decay is that the threads work in parallel allowing them to complete part of the task at the same time instead of one thread working on its own.
Getting more threads to work on smaller parts of the image means that each one has less work to do but it must wait longer for the previous one to finish. This causes a massive performance boost with a few extra threads compared one and small improvements with many of them as the two balance each other out.
The lab machine has 6 physical and 12 logical so you wouldn't expect to see much improvement after 12 threads. If you were using OS threads it would be wasteful to use 16 and you could see an increase in runtime. The part of our code that allows this to happen is the distributor which tasks the worker threads to work on different parts of the image.

## Distributed

Once the Parallel implementation was completed we moved onto the Distributed part. Here decided to move all incrementation for Game of Life to an online AWS server then communicate to this via RPC to establish a connection and output the pgm image.




We moved the Serial GoL logic to a GolServer which the distributor connects to. We achieved this by implementing a single RPC call to process all requested turns. We did this by creating a Stubs.go file which contained all the RPC method calls and the Request and Response values.

There are multiple places where we send and receive data. In the distributor, a call is made to the broker from reportAlive, GetWorld and keypress. ReportAlive sends an empty request (no data) to the broker in order to receive a slice of the alive cells and the current turn. GetWorld sends the initial board state and the parameters (turns, threads, height and width) and receives the final state after the turns. Finally keypress sends a call for each different key. We'll go through these when we talk about the keypresses.

In the Broker, we send a call for the worker (GolServer). These are a small slice of the world, the heights as integers and whether we need to wrap it. The worker will respond with the incremented world. The broker also send a kill command to all of the workers when K is pressed.

The distributor works by using 2 go routines, one in order to manage reporting alive cells and one to manage

keypresses both similar to parallel. The reportAlive function will then send the current turn and the alive cells received from the broker down the events channel. For keypress, if a P is detected, we will send a call to the broker in order to pause the activity. We then send an event saying it is paused, and wait until another P is detected, to resume the activity. It'll then send a message to the broker telling it to resume and send an event saying executing. Q calls the broker saying its quitting, S calls the broker in order to get the board and the current turn and then saves it as pgm image. K sends a call to the broker in order to quit the program.

The broker receives the above from the distributor. When it receives an increment request, it will start sending calls to the GolServer as specified before. When report is called we will return the number of alive cells and the current turn. If P is called we will stop incrementing the board until we receive a second call. If Q is pressed we will send message down a channel in order to return the state of the board to the distributor and stop further incrementation. When K is called it will stop further incrementation like Q, then call all the workers in order to stop and kills the program. If S is pressed it will return the current state of the world.

The GolServer starts of by initialising by a listener, they then call the broker with their address in order to subscribe. When the broker receives this it will dial the server and add it to the list of workers. The server also will receive a call with the world and increment it.

In order to benchmark our system we ran it on local host on the lab machine for each image size for a 1000 turns.

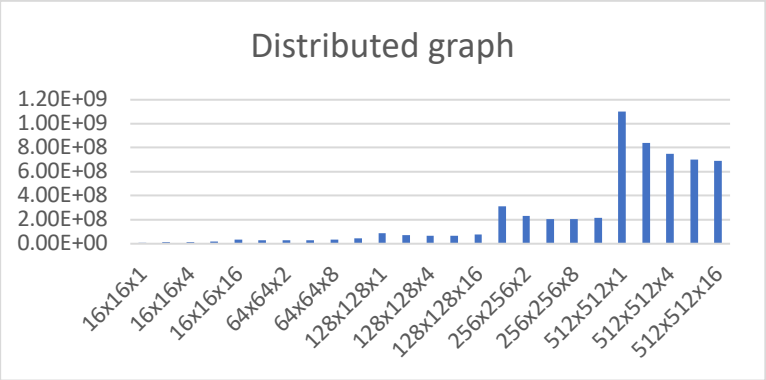



*Figure 3: Runtime of distributed for different image sizes 10 trials*

As we can see from Figure 3 the more workers we had, the faster the runtime, especially for the larger images. When used for larger images the sending smaller slices across the network outweighs sending less network calls, which causes faster runtime. However for smaller images we see an increase in time for more workers due to unnecessary overhead of sending the extra data over the network.

We also implemented fault tolerance so that when a worker dies the broker will reallocate the work to the rest of the workers so that the program can continue running. If the broker were to die however it would cause an error in the program. What this also allows is dynamically increasing the number of workers while the program is running.

## Conclusion

We benchmarked all the different implementations for a 512x512 image for a 1000 turns, and for the parallel and distributed we used 16 workers.

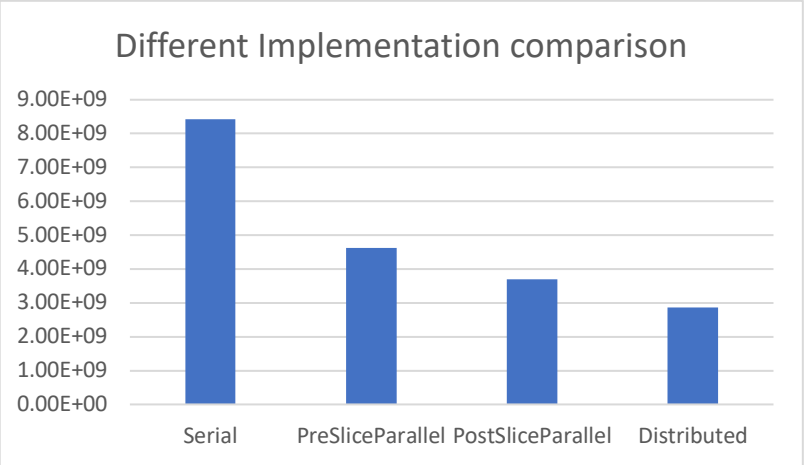



*Figure 4: Comparison of different implementations 10 trials*

From Figure 4 we can see the vastly decreased runtime by splitting the work and running it concurrently.

| | |
|---|---|
| PreSliceParallel | 1.82x |
| PostSliceParallel | 2.28x |
| Distributed | 2.94x |

This table shows how much faster each implementation is compared to the serial one.

From completing this coursework we have seen how varying ways of running programs concurrently can dramatically change the runtime.