**Sengul Cagdal**
**Rahat Mittal**

# Stage 1 – Parallel Implementation

## I. Functionality and Design

### I. i Functionality Implemented

In the initial steps of this project, we implemented a single-threaded execution of Conway's Game of Life. Our program took in a PGM image and transformed it into a 2-D slice of bytes, which represented the state of the game world after a specified number of turns. We proceeded with the parallelization of our sequential method, simply by dividing the height of the image with the number of threads, and having that value be the height of each worker slice. Whereas the width always remained the same. We found that this method resulted in a significantly large slice for the last worker thread, and so moved to modulo operations in the calculation of worker slice heights. We handled communication between the local controller and the workers solely with channels.

For the move from our basic serial implementation to an initial parallel implementation, we parallelized the task of calculating the next state of the world. We used the number of threads in combination with the dimensions of the input image to calculate the way in which the work was split up between the worker threads. For future reference, we would like the record to show that all testing was done on images with their height equal to their width, and that the number of threads are equal to the number of workers.

With the move to parallelism, we've also fully implemented the *keyPresses*. We have exercised the usage of goroutines. Our implementation of the *keyPresses* on the SDL window entails: *'s'* to save the current state of the image; *'q'* to quit the execution of Game of Life on the input image after having saved and output the current state; and *'p'* to pause the execution. When the SDL window is in a paused state, the user can still save the current state of the image (at its paused state) or quit the program entirely. In order to implement this behavior, we've added a *select* statement within the main *for* loop that spans over the number of turns passed into the program. The *keyPresses* case within the select statement, furthermore, entails a nested switch case within it, which switches over each possible key press and executes the correct functionality when thereby triggered.

Following Figure 1, we can observe how the distributor function communicates with the io and SDL components, while also splitting up the

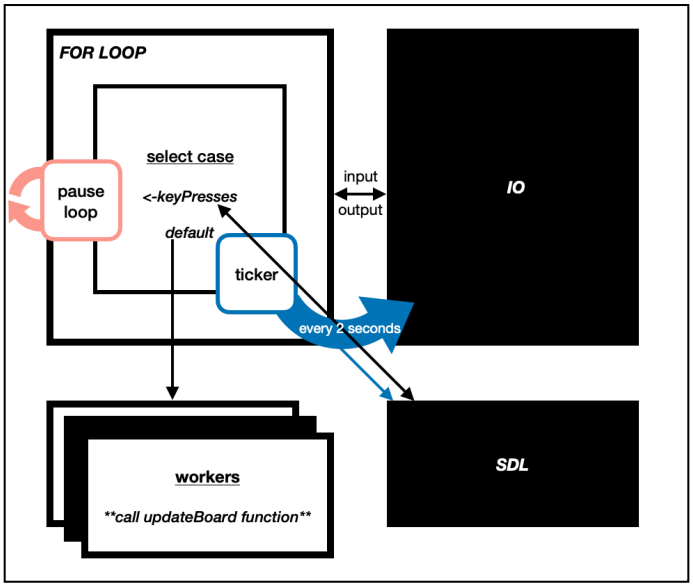



*Figure 1. Visualization of parallel system interaction*

work to the worker threads. The way in which we've utilized goroutines is in the *'p'* *keyPress* case, wherein we must pause execution, and wait until the user enters another key press. This is done by launching the *pauseLoop* function as a goroutine. This function has a for loop which waits on receiving a value through the *keyPresses* channel, and when it does just that, it executes the appropriate functionality. Importantly, this loop only breaks when either a *'p'* or *'q'* is sent down the *keyPresses* channel. Otherwise, i.e., the *'s'* case, it just saves the current paused state of the image and continues waiting for a *'p'* or a *'q'*, as it is yet to be "unpaused".

### I. ii Core Parallel Logic

We have a distributor function which is given the initial world PGM. It takes the image and distributes the image evenly and calls the worker functions as go routines to work on each individual part.Each worker calls the *updateBoard* which implies that there are multiple calls of *updateBoard* running in parallel since *worker* is a Goroutine. each call to updateBoard has a different start and end, thereby defining the slice on which the function is made to work. At the start of the function, we initialize a new 2D slice called *worldOut* with all cells set to 0, this is the slice which the function will eventually modify and return. We use nested *for* loops coupled with *else-if* statements to get the number of alive neighbors for each cell, which is then used to evolve the value of the cell. Once this process is repeated for every element in the slice, we have completed one turn or evolution of the given slice. Due to the wrap around nature of the Game of Life, the function is given access to the entire world as to access its neighbours that aren't contained within the worker's slice. For example, if we are seeing the neighbors of an element at coordinate (0, 0) in a 16x16 world with 2 workers, we would need access to elements: (16, 16), (0, 16), (16, 0) etc. But the slice that is sent to

the first worker would range from (0, 0) to (8, 8) due to which the function needs access to the entire world as just having this slice isn't enough to calculate alive neighbors. After this processing is done, updateBoard returns the slice *worldOut* which is then sent down the *out* channel in the *worker* and is eventually used to reassemble the world in the function *distributor*.

### I. iii Threading Designs and their Analyses

In our first parallel implementation of the game of life we split the image into slices using an algorithm in which we make a variable *WorkerHeight* which is set to image height ÷ threads. For every worker apart from the last worker we pass them a slice of the height *WorkerHeight*. The last worker is passed a slice of height equal to the slice of the image that has not yet been worked on by any of the previous workers.

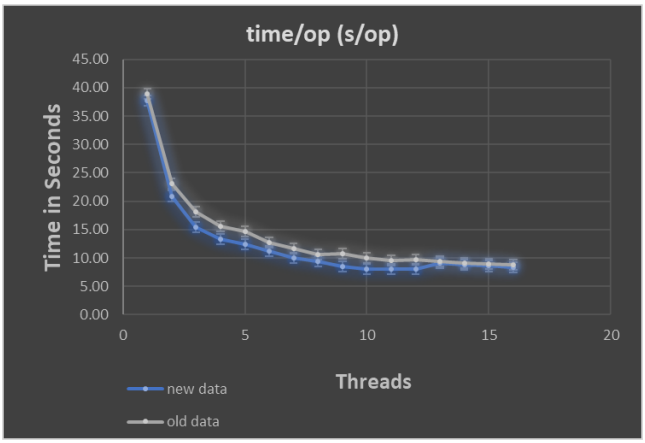
The aforementioned method was found to be rather inefficient in certain cases like when number of threads was greater than image-height ÷ 2. Let us take the simple case of a 16px by 16px image with 9 threads. In this case, *WorkerHeight* would have the value 1. The first eight threads would work on slices of height 1 and the last thread (or the ninth thread) would work on a slice of height 8. Meaning that only one-half of the image was being processed by eight out of 9 threads! This left the other half of the image to be processed by just one thread. We noticed that this method of threading was inefficient hence we decided to think of a new way to split up work amongst threads.

The final algorithm we decided on works in the following manner: we make two variables, one called *SmallHeight* and another called *BigHeight;* where *SmallHeight* is defined in the same way as *WorkerHeight* and *BigHeight* is equal to *SmallHeight* + 1. We send slices of height: *BigHeight, (*image-height % threads) number of times. We also added a counter to the algorithm to see how many slices of height *BigHeight* we send, using which we obtain a start index for where we can start sending slices of height: *SmallHeight* to each of the remaining workers. By the end of this process, we divide the input PMG evenly among the workers.

If this algorithm is used to split up work among threads, then the difference in the height of slices worked on by each thread is at most 1 which means we split up the image in a very even manner compared to the previous algorithm.

## II Critical Analysis of Threading Designs

Observing Graph 1, we can clearly see that the second threading design is faster than our previous implementation, especially for 8-12 threads. For example, in the case of a 512x512 image with 9 workers, run with our old logic, the first 8 workers are sent slices of height 56 whereas the last worker is sent a slice of height 64, implying that one worker must undertake 15% more work than the other workers. If we run the same case with our final implementation, we send the first 8 workers slices of height 57 and the last worker a slice of height 56. Hence, the final implementation is better at splitting up the image more evenly which is likely to be one of the key reasons why it is faster than the previous version. Logically, both the implementations are the same for cases where the image-height is divisible by the number of threads. Such cases are the only time when the initial implementation can split up work as evenly as our final implementation. In events like this the new threading behaves in the following manner, variable BigHeight is never , because image-height%threads is 0, used but instead we just sends all workers slices of height SmallHeight, which is the same as our old threading, as it sent slices of height WorkerHeight to each worker and the definitions of WorkerHeight and SmallHeight are the same.For all other cases, our final implementation splits up the image in a significantly more optimized fashion. Due to this, it is consistently faster than the older implementation. Thus, we strongly feel as though this is a clear indication of optimization of core-logic and analysis from the start of the coursework till the end.




***Graph 1. Comparing previous and current designs of threading.***

### II. i Acquiring Results

In order to compare results from our separate implementations of the parallel implementation to check for improvements, we used Go benchmarking. We gradually increased the

number of threads and measured the total runtimes of the Game of Life executions. Each test used a 512x512 PGM input image, to be processed on 1000 turns and, we took 5 readings to reduce random error. These measurements were taken on university Linux lab machines, which at the moment of writing this report had a 6 core Intel i7-8700 CPU boasting 12 threads.

## III Problems Solved

Throughout our implementation of the parallel program, we've encountered many problems, some of which have been outlined by the aforementioned threading optimizations. But on top of changes to improve the efficiency of our program we've also encountered significant issues with SDL visualizations, i.e., the passing of *CellFlipped* and *TurnComplete* events.

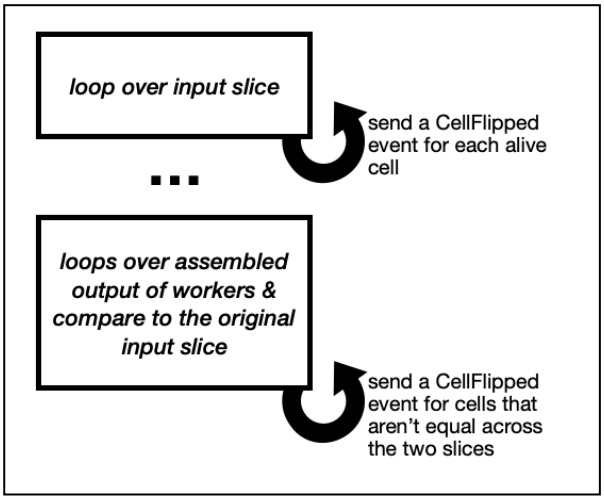
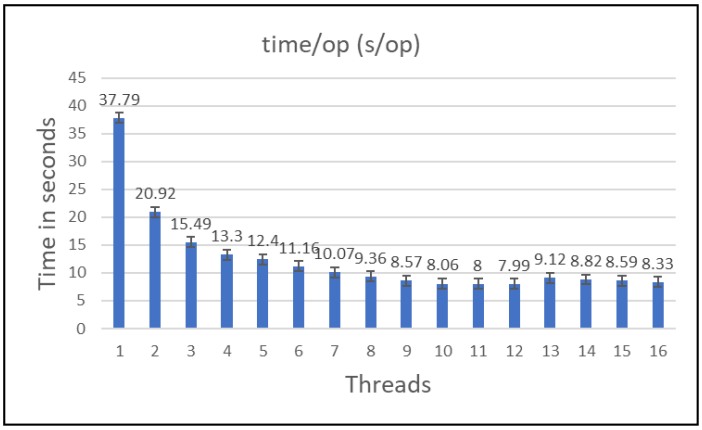



*Figure 2. The implementation of CellFlipped events.*

We found that, at one point, our program was passing all the relevant tests, yet failing on the visualizations of SDL, particularly, it was stuck in an initial *loop* of visualizations, unable to further the execution of the program visually (i.e., in the SDL window, however doing so perfectly fine in the core logic of our code). We solved this problem by only having two moments where we send *CellFlipped* events: once when the image is initially loaded in (to mark the alive cells on the SDL window); and the second when all the workers have finished executing their slices of the image for the given turn, comparing each cell of the initial input and the eventual output.

## IV Conclusion and Future Improvements

We have kept a consistent effort to limit any overheads and prioritize optimization in most of our code, i.e. defining helper functions for any repetitive code sections, only passing the channels necessary to the aforementioned helper functions etc. However, one big oversight in our implementation, mostly due to the lack of time to expand on our parallel code, would have to be the immense overheads that come with channels. If given the time to expand upon our code, we would've undertaken the challenge of using a purely memory sharing design in our communication with the workers. Thus, introducing traditional synchronization mechanisms (mutexes, semaphores and condition variables) to our code as opposed to channels.




***Graph 2. Final readings of the benchmark for 512px-by-512px image run for 1000 turns.***

# Stage 2 – Distributed Implementation

## I Functionality and Design

### I. i Functionality Implemented

To make the first step in our distributed implementation, we followed a simple client-server model, with our local controller (distributor) acting as the client, and a single AWS node acting as the server side. All Game of Life logic was exclusively located on the server side, which will also be referred to as the *worker* henceforth (as we only have a single threaded program across all versions of our distributed implementation, and so a single worker). In the

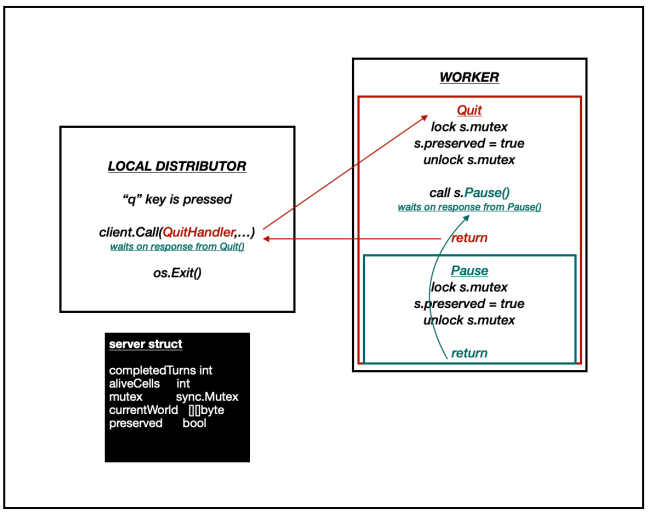



*Figure 3. The implementation of the quit keyPress with fault tolerance*

distributor file, we define structs that are shared between the distributor and the worker, primarily the *gol.Response* and *gol.Request* structs. The distributor function still retains the same nested for-select-switch statements that we had in our parallel implementation. However, instead of executing any Game of Life logic, locally, on the distributor, it passes the relevant data to the worker by making the appropriate method calls. This is where *Remote Procedural Calls* come into play. We have made use of both asynchronous and synchronous calls to the server, via *client.Go* and *client.Call* respectively.

Our latest version of the distributed implementation includes fault tolerance. The way in which we have implemented fault tolerance allows for the local distributor to connect to the worker, and at any given moment quit the local controller in a way that pauses the current worker execution, allowing for a new local controller to connect to the server and pick up where the previous client left off.

The implementation of the goCall method as an asynchronous method call is a key element of our distributed system as it allows for both the Game of Life execution to run on the server while the local controller waits on and handles any keyPresses and ticker events.

### I. ii Overheads

The aforementioned structs (*gol.Response* and *gol.Request)* are utilized in the communication between the local distributor and the worker. The *gol.Request* struct merely consists of a 2D World slice and a variable P to store the parameters (type Params) of the client-defined Game of Life specifications (e.g. image dimensions). Whereas the *gol.Response* struct has a wider array of possible fields, although it must be noted that not all are passed for each and every remote procedural call. The *gol.Response* entails the following fields: 2D *World* slice, a list of alive cells *AliveCells*, the length of that list in *AliveCellCount*, the number of turns the worker has currently executed *CompletedTurns*, and a boolean flag named *Preserved*.

### I. iii Fault Tolerance

(Refer to Figure 4 attached at the end of the report) The field of *Preserved* in the *gol.Response* struct is utilized in our implementation of fault tolerance. The way in which we achieve fault tolerance is outlined in the following diagram.

A key problem faced when we first tried to introduce fault tolerance to our distributed component was that once the initial controller had quit and another had restarted, the initial's Game of Life executions were still running on the server-side. That is to say, the execution wasn't being paused correctly, this also allowed for a data race through the fact that once the second controller had connected and called for execution on the server, it was a race between the two to see which one would be taken as the current world state, e.g., when saving the current state of the world via the key press 's'. We eliminated the bug by calling our pre-written Pause method, as illustrated by the diagram below.

### I. iv SDL Visualizations

For a further extension, we picked up the SDL visualization on the distributed component, however, didn't have enough time to implement it correctly. Our thought process behind this extension was to, firstly, add three new fields: *previousWorld*, a 2D slice; *cellsToBeFlipped* a slice of *Cells*; and *currentTurn* a boolean, to the server struct *UpdateOperations{}*. We, then added further functionality into our main *Update* method (that has the single-threaded Game of Life logic), that would update these values appropriately at the end of every turn. *currentTurn*, particularly, was a boolean flag to mark the start and end of the currentTurn: true being the end, and false being the beginning. We, then, added a new method *FetchSDLData*, which is asynchronously called by the local distributor right after it has called the main *Update* method (or *Continue* to continue the execution of a previously preserved execution, fault tolerance). This method, on the server-side, launched a for loop that "fetched" the required data (*CellsToBeFlipped, CompletedTurns*) and passed it to the method's response *just* if the *currentTurn* value had been set to true. Thereafter, the *.Done* channel of the *FetchSDLData* call was added to the select statement, and in the case that the method had returned, the local distributor looped over the range of the *CellsToBeFlipped* and passed appropriate *CellFlipped* and *TurnComplete* event calls. However, the program neither passed the SDL tests nor had correct SDL window visualizations.

## II Conclusion and Future Improvements

A strong point of weakness in our distributed system would have to be that it solely contains a single-threaded execution of Game of Life, when with the existence of worker nodes, we could've further distributed work, i.e., a parallel distributed implementation. This is shown in our single-threaded benchmarking for the distributed system, which took an average of 59.1 seconds, to process a 512x512 image executed for a 1000 turns of the Game of Life. This benchmark was conducted on a personal computer with an Intel i5-5287U with 2 cores and 4 threads. The system would be sure to scale in terms of connections to make and methods to forward, however this

would be a debatably small price to pay for increased efficiency in overheads and overall optimizations. That mentioned, we did attempt a single-threaded broker solution which would've provided for a great starting point for both parallel distribution as well as the utilization of multiple AWS nodes in some other fashion. However, with the time constraints and our focuses on other extensions, we failed to correctly implement the broker. However, all things considered, we believe we have made a very strong start at a distributed system that can both be scaled in terms of utility, but also optimized in terms of efficiency.


## Acknowledgements

We would like to express our thanks to each and every staff member involved in the unit for their never-ending support. We believe we have learned a great amount of knowledge on both Golang as a language and its core concepts of concurrency as well as its ease of implementation with distributed systems.

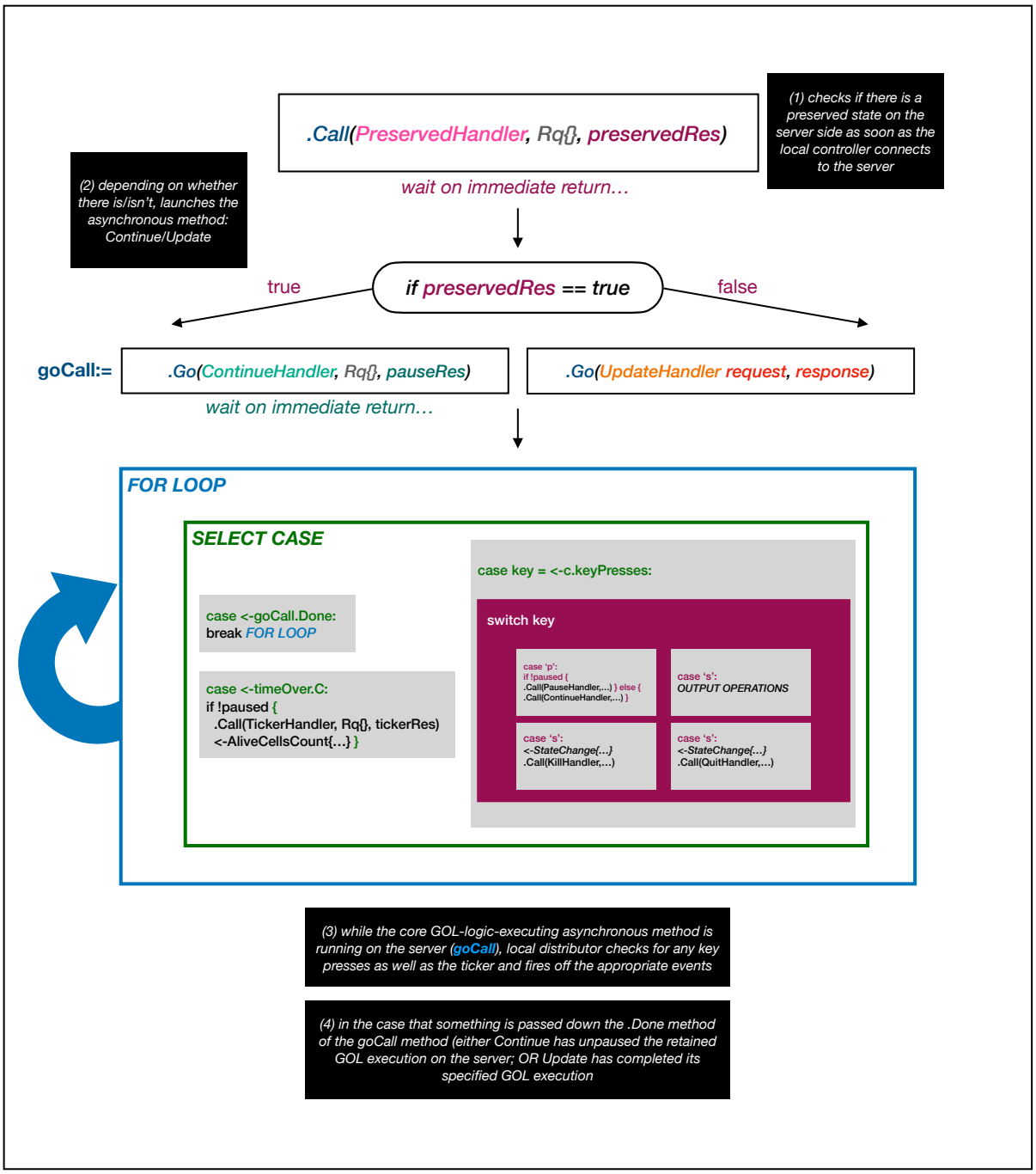




*Figure 4. Visualisation of our implementation of fault tolerance on the distributed system*