Stage 1 – Parallel Implementation

By Rohak Roy(em20413) and Stefan Corneci(xl20294)

Functionality and Design

In the initial stages of development of the program, we started off by trying to get a single-threaded implementation. To do this, we first initialized some relevant channels in order to manage all the IO operations. This was done in the Run function in the gol.go file, which then passes these channels by reference into the distributor function in the distributor.go file. Now that the distributor function has access to all these channels, we wanted to make sure all the IO are dealt with inside here.

This is exactly what we have done. The distributor function takes input from the user in the form of a 2D slice which represents the initial state of the world containing information about all the pixels. We then signal our command channel using the enumerations ioInput or ioOutput depending on what type of operation we wish to execute, i.e., read or write respectively. We also use the Params structure provided in the skeleton code to find the size of the initial world, meaning its height and width. We then construct the filename using this and send it into the filename channel. All this is done to ensure the IO operations in io. go such as readPGMImage or writePGMImage can extract the necessary data successfully execute.

We also created a separate file called helpers. go which consisted of functions that we use repeatedly and multiple number of times throughout the program. These functions were “calculateNextState”, “calculateNeighbours”, and “cacluclateAliveCells”. In brief, calculateNextState simply takes in a 2D world and evaluates it by iterating through all the indexes and then updates each cell depending on its value and following the rules of the game of life. By doing this, we can calculate the next state of the world and return this new world and update our current world to this new one. “calculateAliveCells” simply takes a 2D world as an argument and returns a list of all the cells which are alive in that world. “calculateNeighbours” takes in a world and a x and y coordinate as an argument which is used to identify a cell in that world, it then returns the number of alive cells that surrounding it.

Now that our basic serial implementation works, we further progressed into attempting to make our program concurrent. To do so, we first got the number of threads as desired by the user taken from input. We created as many channels of 2D slices as the number of threads and also split the initial world into as many parts as the number of threads. We then launched a worker goroutine that evaluated each of these smaller parts of the world and then sent the next state of that particular part into one of the channels. At the end of each turn, all these channels containing the separate parts of the world were recombined and merged together in order to reform the newly updated world. Our current world was then updated to this new

world, and we repeated this same process for every single turn until all the turns specified by the user for the game to run had been completed.

The last step was to implement key presses such that we could make run-time changes such as saving the current world by outputting a PGM image of it, pausing and resuming our program, or simply just quitting our program. All this would be done by pressing the letters s, p, and q respectively on the keyboard. Upon pressing a key, the relevant data would be sent into the keyPresses channel. All we had to do is forward this keyPresses channel as an argument into the distributor function and then extract this data by receiving it from the other end of the channel. Once received, we made a simple switch statement to find what exact key was pressed and then react accordingly such that we could see what was being done on the SDL display window itself and the terminal as well using print statement.

It is also worth mentioning we made heavy use of the events channel throughout the code in order to notify the testing functions of any changes made to the state of the world by sending in the relevant events and information along with them. Such events included FinalTurnComplete, CellFlipped, ImageOutputComplete, etc.

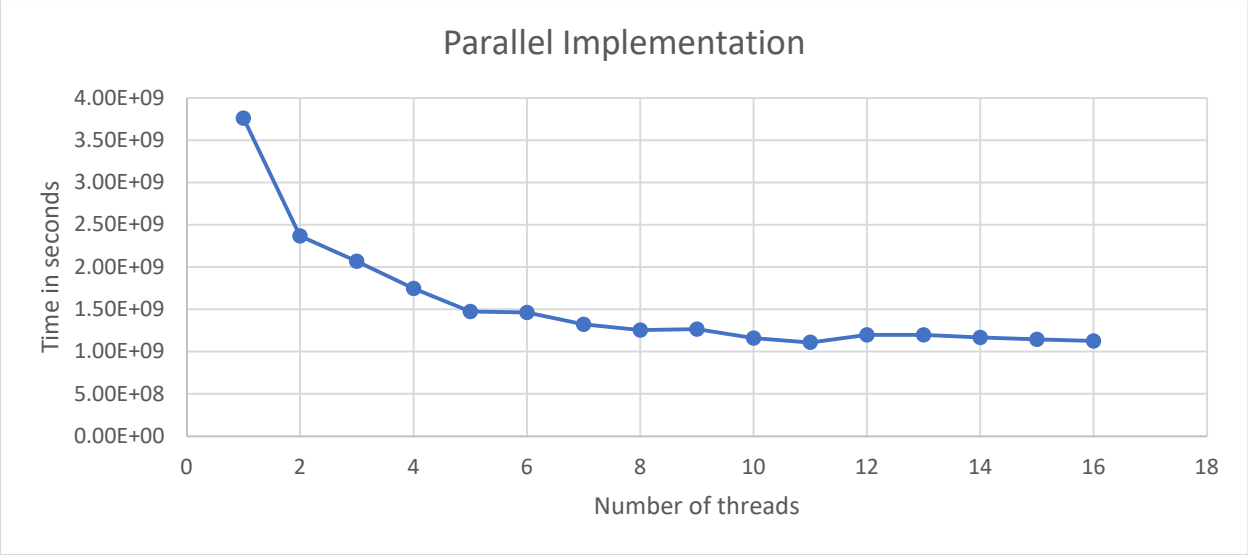
Problems Solved

Due to the fact that our ticker function is a goroutine, this means it was running simultaneously with the rest of our code. Both of these parts were also trying to access and change some shared variables such as the current state of the world and the current number of turns executed. Our code would still pass the test sometimes, but during other times it would fail. We came to the conclusion this was happening because of an attempt to access the same area in memory by two different threads, hence causing a data race. This was one of the major problems we faced during development of this program. The command line instruction “-race” was very useful in debugging our code and figuring out where exactly in it the data race was occurring. Once we managed to locate the area of problem, we used mutex locks in order to solve this issue by making sure only one thread had access to memory of the shared variables at a given time.

Another time-consuming bug we had to find and overcome was deadlocks. Sometimes we would have a channel that would keep on waiting for an input for an infinite amount of time causing the execution of our program to halt indefinitely during run-time. The step over feature of GoLand IDE’s debugger helped us with locating what exactly was causing the deadlock and allowed us to take relevant action in order to fix it.

Testing and Critical Analysis

This parallel method of divide and conquer highly increased as the effectiveness of the program. The time taken to execute it reduces drastically as the number of worker threads increased. This can be visualized using the plotted graph below upon benchmarking our code.



As we can see, the line of best fit in this graph would have a negative gradient which shows that the number of threads and the time taken to execute the program are inversely proportional.

Goroutines are an extremely useful feature of the Go language which heavily aided us with parallising our code. It helped us launch our various worker functions and have them all execute their own part which consequently prevented our code from having potentially long waiting times and allowed it to be independent from any another section within the code. Not having to wait for calls to functions to return anything and just being able to directly move on to executing the next line of code after a call while the functions still run parallelly make concurrent programming incredibly convenient and we believe to have taken full advantage of this.

Stage 2 – Distributed Implementation

Functionality and Design

In this section of our distributed implementation, we had to create a networked system using tcp connections such that we have a client and a server, where the client deals with receiving input from the user and passes all the necessary information to the server. The server then makes use of this information and reports back to client after processing all the data.

In our case, our distributor.go file behaved as the client making a connection to the server whose code and logic is saved in the server.go file. In a holistic view, the distributor forwards information such as the initial world that is read in and the number of turns to be executed to the server, the server then executes the rules of Game of Life for all the turns whilst

updating the world after each turn, and then returns the final state of the world back to the client upon completing all the turns.

Similar to the parallel implementation, my distributer.go , i.e., my client is fully responsible for dealing with the input and output operations. Firstly, it secures an RPC connection to the IP address provided for the server. It then reads in the 2D world initially as passed in by the user. This world along with all its relevant data is then provided to the server. We have done this with the help of a function called “makeCall” which simply initializes the parameters of our “Request” struct using the data extracted from the previously mentioned world received from input. It then creates a pointer to a new “Request” struct. Finally, the client uses this “Request” and “Response” to make a remote procedure call to a function in the server called “ProcessTurns”. This “ProcessTurns” function is simply responsible for actually executing all the turns of the game and updating the world. After the final turn has been completed, it updates the world and turn parameters of our “Response” struct and then returns. The client then uses this updated “Response” parameters to send data back to the IO such as printing out the world after all the turns have been finished and sending events to notify the testing framework.

The other important part of our implementation included calculating the number of alive cells in the current world and reporting it back to the client every 2 seconds. This was done using a similar fashion to our parallel implementation of this. We initialised a ticker than returns every 2 seconds and used a goroutine with a simple select statement to find out when the ticker ticks. Every time the 2 seconds are over, we make use of another function called “makeCallForAliveCells” which once again initialises our “Request” and “Response” struct and then makes a remote procedure call to a function in the server called “AliveCellsFinder”. Our server has two global variables called aliveCells and turns. In the server, every time “ProcessTurns” updates the current world to a new world and increments the turn, the new value of the number of alive cells in this new world and the turn completed are assigned to the global variables. This means the global variables will always stay updated with the latest and most recent number of alive cells in the world and the turn completed. The “AliveCellsFinder” function simply assigns the AliveCells and Turns parameters of the “Response” struct to the global variables mentioned and then returns to the client. The client then uses these values to pass in an AliveCellsCount event to the events channel, hence notifying the testing framework of the number of alive cells every 2 seconds.

Problems Solved

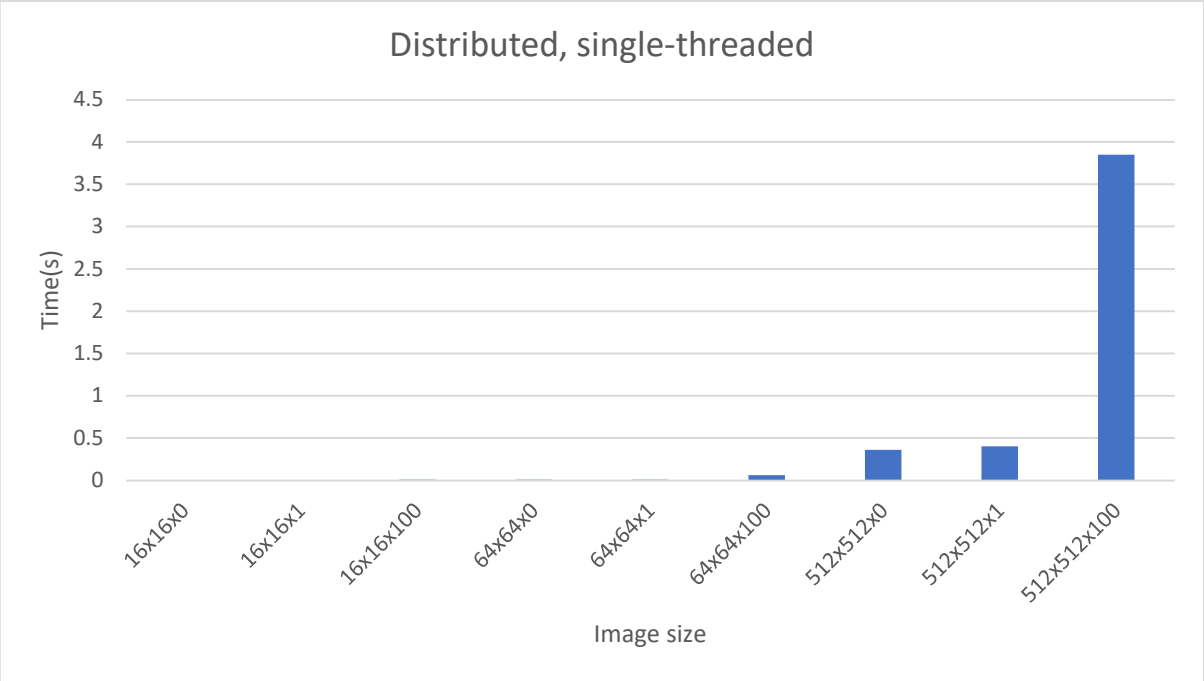
The main problem we faced during our implementation of this part were because of limitations presented by the Go language. Our program sometimes kept on hanging due to data races. Even after much struggling, we could not manage to figure out what was causing it as there were no actual compile or run-time errors. After continuous attempts to fix the bug, upon a lot of researching we found out that Go does not like to make more than one RPC call to the server using the same client. We hence had to stop passing our client value and

instead had to pass it as a pointer using its memory address as a reference. This fixed the data race issue and got our program to function successfully in a consistent basis.

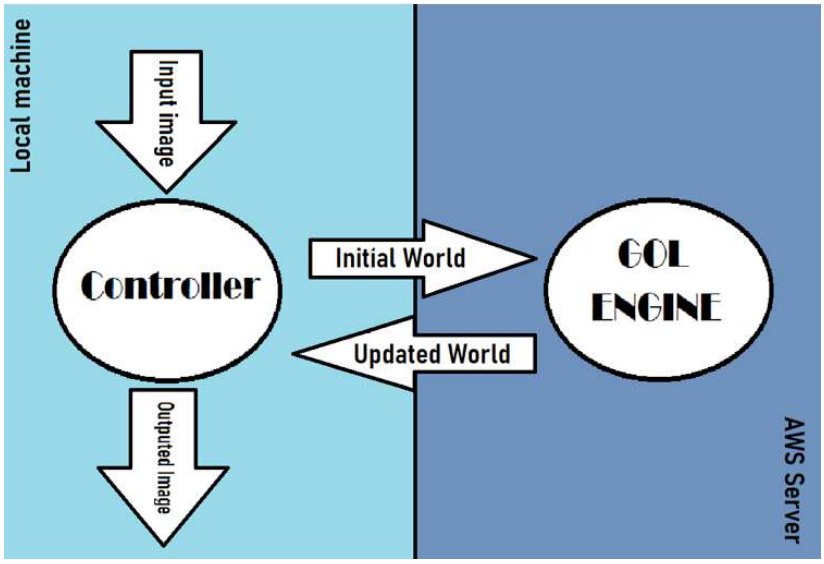
We also had some other data races happening in the server initially, but quickly used our experience from the parallel stage to implement mutex locks and prevent the races.

Testing and Critical Analysis

To conduct our benchmarks, we used c4.xLarge instances on AWS. The graph below demonstrates the performance of our application between the client and server and how the time taken to execute the program varies depending on the size of the image.



Below is a diagram demonstrating an overview of our program for the distributed implementation works



Potential Improvements

The biggest possible improvement would be to implement a broker system that acts as an intermediate between the client and the server. This would reduce direct coupling between the client and server and make our program more desirable. We are fairly confident we would have been able to implement this given a bit more time.

Next time we would also read up on any limitations related to Go so we can prevent from being stuck at a specific step for multiple days trying to find a bug in the code.