## Ben Browne nm20529, Radmehr GhassabtabarShiadeh we20153

### *Parallel implementation*

***Implemented functionalities***

In the early stages of the development did not implement any parallelisation or concurrency practices. To begin with, we focused on implementing the logic of GOL, which involved utilizing a single threaded implementation to take a .pgm file as input, transforming them into a slice of bytes (height and width) and evolving the GoL. The “*gameOfLife*” function sets the ‘*world’* ([][]byte), uses other supporting functions such as “*calculateNextState*” to employ the logic and returns the new world. Once this step was done, we started to parallelise the program where processing the image was executed by different workers. These workers manipulated different parts of the image in parallel to evolve the world and boost run time. To make sure that work is divided evenly, images were always divided by the “*ratio*” of image height over the number of threads. “*worker*” processes the GOL logic while “*controller*” creates the required number of threads and splits up the world. “*sendState*” function makes and sends different states of the board while each key being pressed will be processed by “*keyPresses*” using channels. “*distributor*” function produces filenames, initial cellFlipped events and handles the processing of the specified number of turns; whilst making the various go routines such as “*keyPresses*” and “ticker”: responsible for reporting aliveCell Counts and keypresses as input. Within the “*distributor*” function, an initial 2-D slice is populated byte by byte using the ‘c.ioInput’ channel, other various channels associated with the functions/go-routines are established and the final state of GOL is reported with its associated events sent down ‘c.events’.

***Issues faced***

The first issue faced was regarding the single threaded implementation. The developed code was failing to open 512x512 images as an input while not having any issues with other format of images. To fix this, we decided to create/modify the function “*iterationMaker”* which is responsible for receiving each slice from the workers, appending them and sending to the “*temp”* channel. We opted to have two different code blocks executing with respect to the number of threads active. If there was exactly one thread, the zeroth iteration would receive the slice in a local 2-D slice and send it into the “*temp*” channel. In all other cases, we would iterate through the number of available slices received from the worker channels, recreate the “*world*” before sending to “*temp*”; to be received by “controller” and subsequently returned. Another problem encountered was to do with the “*controller*” function. We could not make the function present (and its associated n. threads) throughout the entirety of the GOL’s runtime as we faced errors while trying to utilize different “*waitGroups*” – trying to Add() to a waitgroup whilst another thread was already blocking on Wait(). To overcome this, we decided to call “*controller*” for each turn. The function creates ‘p.Threads’ different go-routines (“workers”) and sends the final slice of the world down to “*temp*” channel before returning. We believe that this solution created a noticeable performance loss when executing our implementation vs having the same specified number of “workers” present throughout the entirety of the program.

### *Parallel part – Analysis and tests carried out*

***Testing method***

The method of testing was using Go Benchmarking. We compared the time taken for the output when different number of threads were active. This was repeated 6 times for each number of threads, on 512x512 and 64x64 images, for 500 turns each, on a 4-core machine (8 virtual) with the average time was reported in seconds.

***Results***

The figures above show the time taken with respect to the number of threads and image type. We expected that parallelisation, the average time taken should be halved as the number of threads increase to handle different parts of the image. The work divided is defined by the “*ratio*”: .

***Analysis***

The chart shows that the series implementation is much slower than its parallelised counterparts and can also deduce that as the number of threads increase the average time taken to produce and output decreases. However, the runtime does not always halve, as one may expect. In the 512x512, we can see that the fastest thread is 8 threads rather than 16. Furthermore, this is also visible when we look at the runtime of 64x64 images. This might be due to the implementation we have decided to go with as we believe that the workers don’t get assigned proportions efficiently. It is also shown that using smaller images, such as the 64x64, decreases the time taken rapidly and further reinforces the statement about the division ratio as the are no improvements after the 8th thread. In general, the time taken did improve as the number of threads increased however the improvement is much more noticeable when upgrading from a purely series implementation. As the number of threads increase, the time improvement is far less significant. This might be since the input and output part runtimes were also included in the tests and as these are not parallelisable may have introduced unseen overhead In the benchmarks. Furthermore, the test machine only has 4 physical cores, meaning that using more than 4 threads results in negligible improvements and the benchmark results demonstrate that since after 8 (due to hyper-threading), the program becomes even slower. This may also be due to a lack of optimisation within Go itself upon runtime, unable to efficiently allocate work and manage an increased number of worker threads.

***CPU Execution Profile***

A picture containing text

Description automatically generated

Above is a picture of the CPU profiling of our implementation’s runtime, created using ‘the powerful tool’ for go. This CPU model demonstrates how much processing power each component of the runtime in this implementation takes. We noticed that input and output (*io.go*) of images along with “*SDL*” take the most processing power in this program. Furthermore, the main runtime required the highest processing power, which included “*SDL pull events*” functions. The workers on the other hand were far less demanding compared to the main runtime of the program. The workers in total took about 36.46s to complete which included calculating the net state of the board and reporting the surroundings of a cell. The whole program took a total of 147 seconds to complete. This test lets us understand that as the number of workers increase the runtime of the program does not half as we expected it to be. This mostly due to the workers taking up only ~16% of the processing power with all the other functions as discussed above along with various system calls, still taking the larger part of the processing power.

***Potential Improvements***

Potentially, we would have improved he divide ratio of the program to remove the overhead of creating and destroying ‘p.Threads’ number of workers every turn a turn was to be executed. Given more time, having multiple different implementations would have allowed us to conduct further analysis into the bottlenecks of our system. One such implementation that would’ve improved our runtime would have been put finished “worker” threads to sleep, collate the new ‘world’, before waking and allocating the workers with updated slices in parallel for processing after every turn.

# *Conclusion*

In conclusion we can observe that the parallel implementation is much faster than a single threaded series program. The graphs also demonstrate how well the program scales as the number of workers increase however that was limited due to computation power of the used machine and the way our system was designed.

# *Distributed Implementation*

***Implemented functionalities and design***

In this step, we used the implemented functionalities from the previous stage to process the logic of GoL. However, the functions and roles were separated into different components for this the problem to setup different remote procedure calls (RPC) between the client and the server. A broker was also introduced in the implementation, responsible for the communication between the server and the client (local controller). The “*stubs.go*” file is used as a template by the server, client, and the broker. The file has all the struct parameters and method call strings used for RPC. “*server.go*” within the “*server*” directory, is responsible for processing all the logic of GoL, whose methods are only ever called by “broker.go”, in order to reduce coupling between our local controller and the GOL workers. “server.go” runs on an AWS node that calculates the next state of the GoL board and “*distributor.go*” is responsible for IO and capturing key presses, being the client on the local machine. Within the server, the next state of the GoL gets calculated via the “*calculateNextState*” function, while function “*controller*” splits up the world and starts the required number of threads. Once all the required logic is processed the new world is returned to the broker via an rpc reply interface – Update{}, which contains the new World and Turn number. The “*distributor*” function in the client has all the same functionalities from the parallel implementation with addition of calls to the main “*broker*” method to grab updated ‘worlds’ and to pause/continue processing of the GOL. Pausing is achieved using the respective ‘Pause’ and ‘Continue’ methods in broker which result in the ongoing main broker method – ‘Broker.Broka’ to enter a loop and not continue processing turns by calling the server. After the broker main method returns our local controller then outputs the final world and sends the needed events. Work was split between different number of threads the same way it was done within the parallel implementation however changes were made to accommodate an RPC connection communication rather than using channels.

All RPC calls require a string, request and a response/reply. The strings correspond to the invoked methods name, the request is a struct containing various data fields (ie ints, [][]byte, Params) which is used to pass data from the caller -> method body. The reply, often passed as new(Reply), enables the caller to receive data back from the method. In our case this was often a Update{} – containing the updated world and turn number. Empty{} can also be used as requests/responses when no data is required to be passed over the connection.

Decoupling the server and local controller via a broker allowed for easier implementation of desired functionality (ie pausing) and the added benefit of not causing errors on the server side; if the local controller was to unexpectedly shutdown: as connection between the broker and server is maintained.

We also decided to implement an SDL live view which was an extension task provided for this stage. This was handled within the local controller, making rpc ‘Update’ calls to the broker every 33ms and sending cellFlipped events via the ‘CellFlipped’ function. 33ms was chosen in order to achieve a targeted 30fps live view of the game, which we deemed satisfactory.

***Issues faced***

One of the issues we faced was at the early stages of implementation., when trying to send a channel through RPC call from the client to the broker. In the client we created ‘iteration’ ([] chan [] [] byte), to be used by the func ‘iterationMaker’ to append the new world together each turn. As explained above, the world is reported back to the broker by the server. So, to fix this issue, we moved the creation of ‘iteration’ to “*server.go*” which fixed this issue. The other issue we fixed was a race condition within our code. We found out that as GoL evolved on the server, and number of alive cells being reported by the server, were not in sync so the wrong numbers were sent back to the local controller. To fix this issue, we further employed a custom mutex lock within the controller. The controller calls the broker to pause, allowing time for the broker to finish updating the world for that turn, before calling ‘Update’ to receive this new world, processing all logic to send an ‘AliveCellsCount’ event and finally telling the broker to continue. This ensures the broker stops executing turns whilst the local controller sends the needed event. The last issue we faced was the button press “k”. this button is tasked with shutting down all components cleanly without any errors. Unfortunately, we could not find a solution to this problem in a reasonable time, and it remained unsolved.

# *Testing method*

Go benchmarking was used again for his stage for both with/out SDL extension task. We set up a t3.Micro instance (most powerful available) of an AWS which has 2 cores rather than the 4 core machines used for earlier (hyper threaded to use 8). We decided to max our benchmarking to 8 working threads because of the virtual CPU power stated above. The instance would run all the workers in “server.go” with the broker and controller ran locally. We decided to use only 512x512 images for 100 turns and repeated the test 6 times to report an average time.

***Results***

As we expected, the benchmarks were far slower than the parallel implementation of this assignment. As we were using only a 2-core AWS node with one instance, the performance does not improve at all and worsens as the number of workers increase. The general worse runtime was expected since RPC calls are more time consuming. In theory, if the core of the machine was unlimited and different instances for the node were created, we could have expected a better runtime as the number of workers increased in a similar fashion to the parallel stage.

***Potential improvements***

To improve the developed code, one potential improvement would be using multiple instances of AWS nodes, with one node responsible for a section of the world. We believe this would improve the runtime of the program when paired with parallelised workers within the nodes. A central system would be needed to ensure all nodes remained in sync with the correct world. However, it should be stated that using RPC calls is far less efficient when it comes to runtime as they take majority of the runtime processing power. The other improvement we could make would be functionality of the program. Our implementation does not correctly process the “k” key press. Implementing this correctly will improve the program for correctness and allow for an easy shutdown of all components simultaneously. Ideally the broker should be ran on a separate machine to the local controller, to create a truly distributed system. This however in practise would significantly further increase running times, making connection speeds/bandwidth an even larger bottleneck.

# Conclusion

Throughout this stage, we saw how inefficient this implementation is compared to the parallelised program is the first stage. The graphs demonstrated how the program scales as the number of threads increase. We also observed how one instance with a low power 2 core virtual CPU paired is not the best way to process RPC connections.