A Parallel and Distributed Implementation of Conway's Game of Life

Chris El Akoury
iy21003@bristol.ac.uk

Emir Abou Zaki
jk22756@bristol.ac.uk

November 2023

1 Introduction

1.1 Game of Life

Conway's Game of Life is a classic cellular automation created by mathematician John Conway in 1970. The game's evolution is completely determined by its initial state, with no further human interference. The Game of Life is played on a two-dimensional grid of cells. Each cell can either be alive or dead. Initially, the grid is populated with an initial pattern of alive and dead cells.

1.2 Project Overview

In this coursework we implemented the different parts of both the parallel and distributed systems of the game of life including any optimisations to be made, passing all the tests provided. We then tested, bench-marked and compared the different implementations and optimisations we have worked on which includes: detecting bottlenecks and optimisations on the parallel implementation, optimising the overhead of the communication between worker nodes by applying a halo exchange scheme, and finally implementing a parallel distributed system.

2 Parallel Implementation

2.1 Serial Implementation

We began with a serial implementation of the Game of Life, which utilizes a method to calculate the next state of the game of life based on the current state of the world, for this to be done correctly, we need to keep track of the current state while applying the rules and producing the next state, this is done by creating a temporary newWorld variable of 2D bytes which is used to store the values of the next state while the original world is used as the current state to check on while computing. The world is also wrap-around bounded meaning the top of the world is connected to bottom and the left wraps around to the right. We were able to solve this in the calculateNumOfLiveNeighbours method by using the modulus operator on the height and width of the world when looking at a cell's neighbours, which properly applies the wrap around functionality. We then tested our serial implementation with the provided tests and moved on to parallelising the Game of Life.

2.2 Parallelising Game of Life

Parallelising the game of life was done in the distributor, which divides the world relative to the number of threads and runs the worker in a goroutine, receives the next state from the workers, and reconstructs the world before sending the next iteration. The distributor initializes a slice of output channels of 2D arrays of bytes and creates a worker channel for each worker within that slice. The distributor also initializes a 2D slice of bytes newWorld. After the initializations, the distributor divides the image to bands of equal size using floor division and in the case where that doesn't divide equally with the number of worker threads, the distributor keeps track of remainder pixels by getting the modulus of the height of the world with the number of worker threads and adds it to the last worker. The distributor runs the workers in a goroutine with the indexes of its band, the world, and that band's output channel. The worker processes the next state (using calculateNextState) and sends the resulting newWorld back to the distributor via it's output channel. Once the distributor receives the data from all of the worker goroutines, it reconstructs the final evolved world. In order to output a PGM file, the distributor calls the outputPGMFile method which interacts with the io channels. This method takes the image parameters, the distributor channels, an integer 'turn', and a 2D slice of bytes 'world' as inputs. The method proceeds to loop through the image byte by byte and sends the cell through the ioOutput

channel at every iteration. Once it has done that, it puts ioOutput in the ioCommand channel, signaling to the io to get ready to generate output. Finally it sends the filename through the ioFilename channel and signals that the image output has been completed.

2.3 Tests and Benchmarks

The tests and benchmarks for the parallel implementation were conducted on a Dell XPS 15 laptop with a 14-core 20-thread 12th Gen Intel(R) Core(TM) i7-1200H processor running Fedora Linux. For accuracy and fairness all the benchmarks and tests were conducted while the machine was connected to a power source (charging) and with no other tasks running in the background. Each benchmark was repeated 6 times and the average runtime in seconds has been recorded. The bar graph in figure 1 shows that our parallel implementation improves in performance and in runtime as the number of worker threads increases. However, the increase in performance reduces as we approach a higher number of threads which could be due to overhead from the number of channels needed to communicate between the worker goroutines and the limitations of the number cores of the processor to run in parallel.

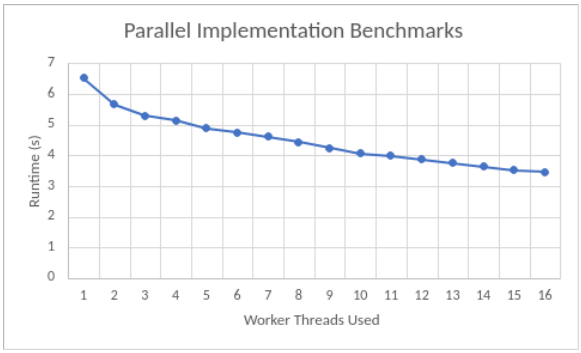

Figure 1: Bar graph showing the variation of runtime (s) as a function of worker threads used of the parallel implementation of GoL on a 512x512 image for 1000 turns.

2.4 CPU Profiling

To better understand how our program works and where it spends most of its time computing. We decided to use a powerful tool pprof to get a profiling of the CPU of our parallel implementation using 8 worker threads for 20 trials. Figure 2 shows the CPU profiling of the initial parallel implementation. We can deduce that the program spends a lot of time computing in the getNumOfLiveNeighbours method, which uses the mod operator to get the index of the wrap-around world. Since the mod operator is an expensive operation and is used multiple times in that method. We decided to optimise this by replacing the mod operator with if statements which handles the cases where the current cell's neighbours indexes need a wrap-around.

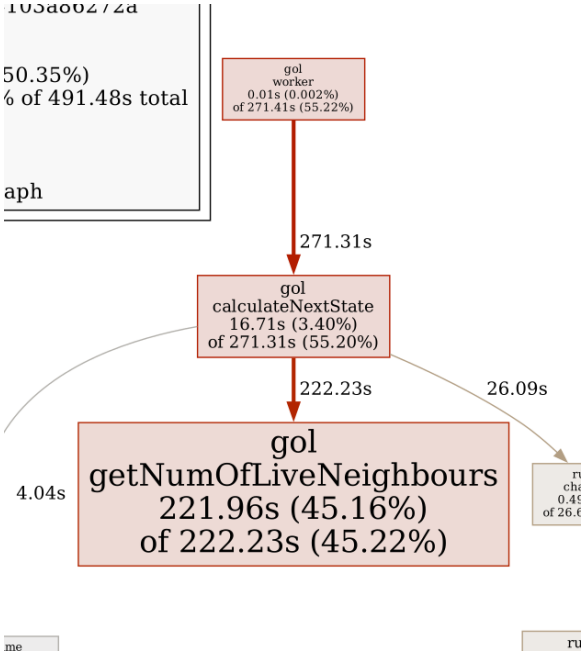

Figure 2: CPU Profiling of Initial Parallel Program using 8 worker threads for 20 trials.

After handling the bottleneck and replacing the modulus operator. Figure 3 shows the CPU profiling of the parallel program after applying the optimisations which shows a decrease of 70% in the time spent in the getNumOfLiveNeighbours method from 221.96s in the initial method using the mod operator to 66.82s after replacing the mod operator with if statements.

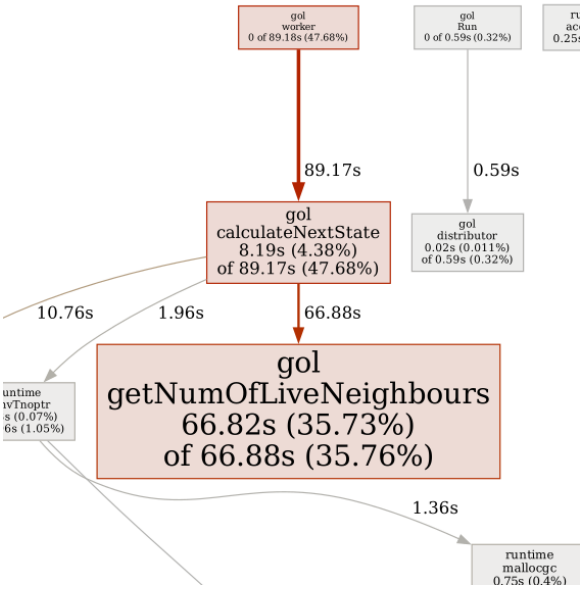

Figure 3: CPU Profiling of Parallel Program without Mod Operator using 8 worker threads for 20 trials.

To further verify this optimisation, we conducted several benchmarks for both the initial parallel implementation and the implementation after optimisation which can be seen in figure 4 that shows the improvement in performance after replacing the mod operator.

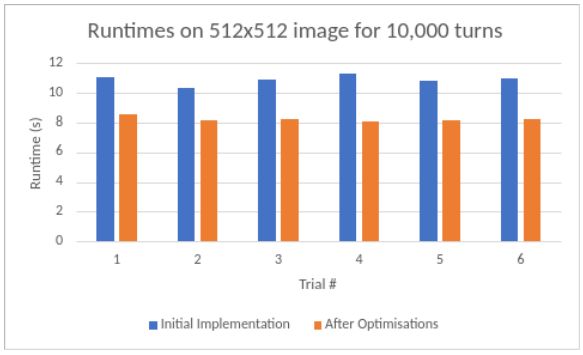

Figure 4: Bar graph comparing the runtimes (s) of the initial parallel implementation and the parallel implementation after optimisations using 8 worker threads.

3 Distributed Implementation

3.1 Functionality and Design Architecture

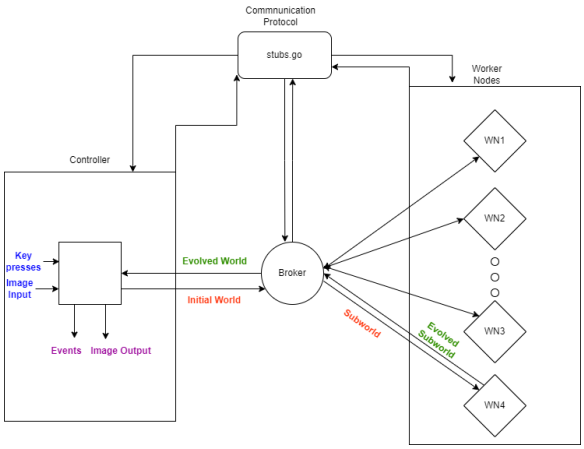

Figure 5: Diagram of the architectural design of the distributed system.

The first part of designing our distributed system was to separate the computation of the game of life logic and the controller to run on different machines and properly communicate with each other. We achieved this by setting up a worker that listens to RPC calls to which the controller connects in order to call certain RPC methods to run the Game of Life with a set of parameters. The protocol of this communication is defined in stubs.go which is shared across the distributed components. After having a serial distributed implementation working, we decided to add a broker node to decouple the controller from the workers. The broker will be responsible of listening to RPC calls from the controller and then distribute the workload and send RPC calls to each worker in the system. This was achieved with scalability in mind such that the broker will be able to handle and work properly with any amount of workers. We designed our broker to be given the worker addresses as a flag and connects to each of them saving a pointer reference to each worker client. The controller starts the game by calling the EvolveGoL RPC method of the broker, the broker then partitions the world depending on the number of workers it is connected to similar to the parallel implementation, where it divides the world using floor division on the number of workers and retrieves the remainder rows using the modulus operator of the height of the world with the number of workers, which will then be added to the last worker's partition. This ensures that the dis-

tributed system is scalable to any number of workers. Figure 5 shows our design of the controller, broker and workers as well as the communications between them.

3.2 Evolution of The Distributed Implementations

The following sections demonstrate the five versions of the distributed system. Each implementation has been bench-marked and tested, and a comparison of their performances against one another as well as their capabilities on the cloud (AWS Instances) were obtained.

3.2.1 v1: Broker Multiple Worker Nodes Implementation

The first version of the distributed system consisted of splitting the work across multiple workers to compute the Game of Life. This was achieved by assigning the subsections of the world in the broker and pass the indexes that needs to be processed by each worker. However, at each iteration, the whole world shall be sent on every RPC call since this implementation is dependant on the entire world and only uses a start and end indexes to process its part.

3.2.2 v2: Halo Exchange with World Reconstruction Implementation

The second version is based on the halo exchange scheme. To process all the cells properly, each worker needs a row from its neighbours called the halo regions. This was implemented by constructing the subworlds alongside their halo regions on the broker's side, then each subworld and their halo regions are sent to each worker to be processed. After each iteration the workers send back their computed subworlds and the broker redistributes the halo regions and reconstructs each subworld to be sent again to process the next iteration. This version is better than the previous where the data sent over the network is reduced to the size of each subworld and their halo regions. However, further improvements can still be made.

3.2.3 v3: Pure Halo Exchange via Broker Implementation

The next optimisation to be made is to implement a pure halo exchange scheme, where only the halo regions are sent over the network without having to send and reconstruct 2D subworlds at every iteration. This was achieved by first sending the subworld partitions to each worker along side their initial halo regions which is received by the workers on its own RPC call. The workers will process the next state and return the halo top and halo bottom back to the broker. Furthermore, the broker will wait for all workers to return their halo regions for the current iteration and then redistribute the halo regions to each worker to process the next iteration and send back the halo regions again. By only sending the halo regions over the network saves a lot of overhead as the size of each halo region is very small compared to the entire world.

3.2.4 v4: Peer to Peer Halo Exchange Implementation

Our next version is implementing a peer to peer halo exchange scheme between the workers, and thus making the computation independent on the broker. Figure 6 shows the design architecture of the peer to peer distributed system. The broker first sends the address and port of the worker to connect to to initialise each worker node. This will determine the communication of the halo exchange between the workers. For this to work properly, at each iteration one of the workers called the startNode will start the halo exchange chain where the worker will send its halo bottom to its right neighbour and will receive the halo top as a response. This goes in a loop of all the workers until it reaches the startNode again where the chain will terminate and the next iteration can be processed. This doesn't differ in performance compared to the previous implementation since the same over head persists, however in situations where the worker nodes are in the same region, this will cause a faster response due to the peer to peer nature of the design.

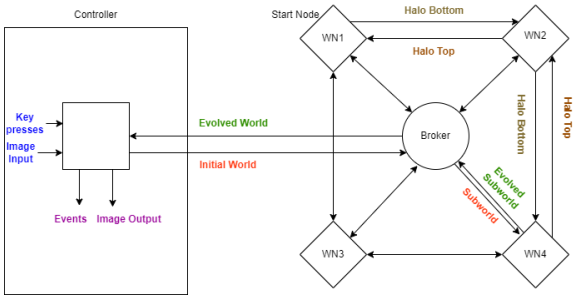

Figure 6: Architecture design diagram of the peer to peer halo exchange implementation.

3.2.5 v5: Parallel Distributed System with Halo Exchange Implementation

The final version of our distributed system is to combine the parallel implementation into the dis-

tributed one. This was a variation of the v3 implementation, since it was our most stable version at the time. To parallelise each worker in the distributed system, we introduced a method runParallelWorkers which accepts the number of worker threads and the halo regions as parameters. The workers are then run in a goroutine given the subworld of the worker and its partitioned indexes of the world excluding the halo regions. After the next state is calculated the worker returns the halo regions to the broker and waits for the new halo regions to process the next state. This implementation takes advantage of the concurrency of the Game of Life in the parallel implementation as well as the its distributive nature across multiple worker nodes, which shows an substantial increase in performance compared to the previous versions of the distributed system.

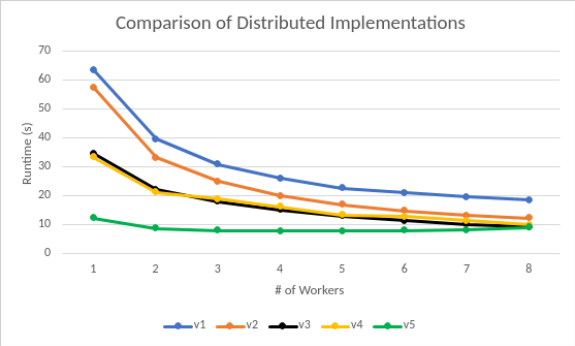

Figure 7: Graph showing the variation of execution time (s) as a function of the number of workers for different implementations of the distributed system.

Figure 8: Bar graph showing the variation of runtime(s) as a function of the different versions of the distributed system on a 5120x5120 image using 4 worker nodes

3.3 Comparison of The Different Implementations

We bench marked each of our distributed implementations and compared the results as shown in figure 7. Similar to the parallel benchmarks, the tests and benchmarks were conducted on a Dell XPS 15 laptop with a 14-core 20-thread 12th Gen Intel(R) Core(TM) i7-1200H processor running Fedora Linux. All the benchmarks and tests were conducted while the machine was connected to a power source (charging) and with no other tasks running in the background. For every number of worker nodes, we ran the benchmarks for 5 trials and recorded the average runtime in seconds. To do so, the main method of the controller was modified to print the execution time after completing a certain number of turns. Figure 7 shows that all of the distributed implementations scale well as the number of workers increase. In other words, as we increase the number of worker nodes, the runtime improves for all the implementations. Moreover, we can observe that every implementation presents an improvement in runtime compared to its previous implementation. This improvement in performance is deduced to be relative to the size of the data sent with each RPC call, except for v5 where the increase in performance is due to the parallelisation of each worker node. Furthermore, the distributed implementations were also tested against a large input image of 5120x5120 with 4 workers for 1000 turns as shown in figure 8, this shows that the performance of the implementations is maintained and shows the exaggeration of the difference in runtime compared to one another.

3.4 AWS Instances Benchmarks

The distributed system was tested on AWS on the t2.micro instances located in North Virginia, USA. To setup these instances, we found it useful to create an image template of an instance that contains the go program, the git repository of the distributed system alongside the required network rules for the tcp port of each worker. Using this image, multiple instances were created to test the distributed system. The controller and broker were setup on the same machine to avoid port forwarding issues, the broker was setup with the addresses and port of each worker instance. Benchmarks were conducted on the v3 and v4 versions of the distributed system on a 512x512 image for 500 turns as shown in figure 9. The bar graph in figure 9 shows an extreme difference in performance between the v3 broker halo exchange implementation and the v4 peer to peer halo exchange implementation. The delay in the v3 implementation is due to the dependency of the broker at every iter-

ation and thus the RPC calls from the broker to each worker took more time to be sent since the broker and the worker instances are in different regions. However, the v4 implementation shows a much faster runtime due its peer to peer nature between worker nodes, and since these nodes are located in the same region, the RPC calls between them is much faster after every iteration in comparison to the v3 implementation.

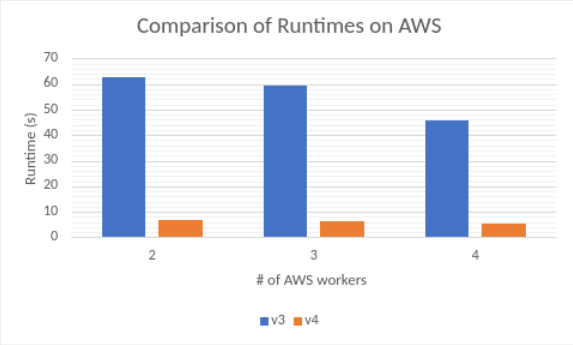

Figure 9: Bar graph showing the variation of runtime(s) for v3 and v4 of the distributed system on 512x512 image for 500 turns

3.5 Potential Improvements

A potential improvement to the distributed system would be making it fault tolerant. Fault tolerance is crucial in large scale distributed systems where the possibility of a failure in a distributed component is high. Handling a failure and disruptions will make the system more stable. When a worker node get disrupted or disconnected, the broker shall be able to adjust and redistribute the work load to the other available workers such that no loss in information occurs. Moreover, in the case where the controller gets disrupted or disconnected, the broker shall be able to keep track of the current state and resume computation once another controller connects again.

4 Conclusion

In conclusion, this project successfully implemented and optimized both the parallel and distributed implementations of Conway's Game of Life. The benchmarks and tests conducted showed scalability in our distributed system, with significant improvements achieved by optimising communication overhead via halo exchange. The integration of the parallel implementation within the distributed system showcased a large improvement in performance. This project has provided us valuable insights into the complexities of parallel and distributed computing, and the importance of good design and architecture choices.