Lokhei Wong uy19297
Pragya Gurung wq19451

Stage 1 – Parallel Implementation

1. Functionality and Design

1.1 Functionality Implemented

Our initial implementation of Game of Life did not include any principles of concurrency or parallelism. It simply involved transforming an initial input of a binary image into a 2D slice of bytes, representing the initial board state. For every turn, the slice, acting as a closed domain, would evolve according to the Game of Life logic. After the board has computed for the specified number of turns, the program would terminate and a PGM image would be given as output.

Having built a basic serial implementation, we then moved onto parallelising the program, where the task of processing each turn was distributed between multiple workers (according to the requested number of threads). Each worker is responsible for an approximately equally sized board segment. Our solution did not use memory sharing; rather, the goroutines communicated by sending and reading from channels.

After implementing the parallelised version of the Game of Life, we reported the number of alive cells every 2 seconds using a ticker goroutine running in parallel to the distributor. The results would be sent to the events channel to be outputted. We then worked on visualising the game state using SDL. In order to aid the SDL visualisation, we added 5 keypresses in the same goroutine as the ticker: ‘s’ to generate a PGM with the current state of the board; ‘p’ to pause and resume the program; ‘q’ to terminate the application; ‘r’ to reset the world to its initial state and finally ‘k’ to set a game state which the board can revert back to. The exact key that was pressed would be received from the *keyPresses* channel that is passed to the distributor function.

1.2 Problems Solved

Using an in-place algorithm to implement the GoL logic would not have been ideal as it could have led to miscommunication between cells. To solve this, we applied the logic to each cell on the board and stored the final world in a new 2D slice.

The board is considered a closed domain. As a result, any cells on the edge of the board would need to interact with the cells on the other side of the board. This restriction was only relevant when calculating the alive/dead neighbours of cells on the edge of the board. We had to find a simple solution to access cells on the opposite edge - this was initially solved using the modulus operator.

Another problem that we came across was a method to split up the board state so that each worker thread would be computing the next evolution on approximately equally sized segments. In particular, we had to figure out a method to split up the board for threads which were not divisible by the size of the boards (for the given test input image sizes, these were the threads which were not a power of 2). Initially, we used floor division to work out a standard segment height, which all but one worker would operate on. The final thread would then take the rest of the board that was left. However, this method is not very efficient as the last thread would sometimes be taking up to double the segment size compared to the other threads, minus one. As a result, we implemented a method to split up the height so that it would only differ by at most one between different worker threads. We did this by assigning to each of the first n workers (where n is the remainder when height is divided by the number of threads) one more row compared to the rest of the workers. The efficiency of these two different threads split implementations will be discussed below in the *Critical Analysis.*

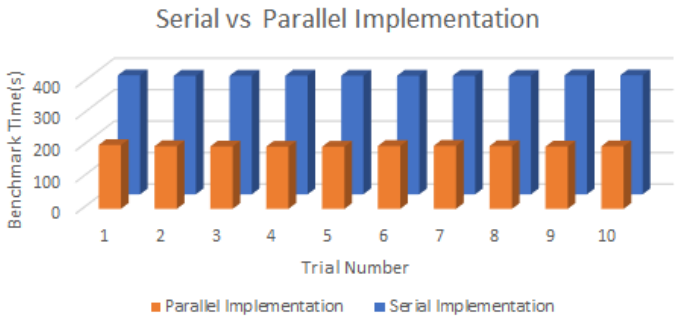
2. Testing and Critical Analysis

2.1 Acquiring Results

All tests were repeated 10 times on a 6-core 12-thread university lab machine which we accessed remotely through x2goclient. As it was unlikely we would be able to connect to the same machine again, all benchmarks were completed on the same day on the same machine to ensure that our results were not affected by which machine we would use.

2.2 Serial Implementation vs Initial Parallel Implementation

Our benchmarking tests produced the following



results for the serial and initial parallel implementation:

Figure 1: Total benchmark time for serial and initial parallel implementation

Serial benchmark - Mean: 377.01s, variance:0.418s, range: 1.743s
Initial parallel benchmark - Mean:178.58s, variance: 2.12s. Range: 5.501s

Our benchmark data yielded a low standard deviation, with all benchmark times clustered around the mean, indicating a low variability between our benchmark tests. As evident in figure 1, our initial parallel implementation scaled a lot better relative to our serial implementation.

2.3 Optimised vs Non-optimised Thread Split

As mentioned previously, after successfully implementing a parallel version of the game of life, we continued to make changes in order to improve the efficiency of our program. One such change was the method in which we split work amongst threads.

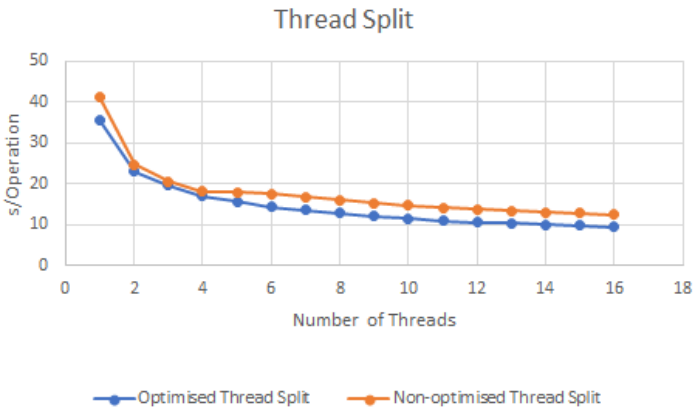


Figure 2: Average time per operation (512x512x1000) over range of threads over 10 trials

Figure 2 compares the time taken per operation between the 2 different methods. As expected, the non-optimised way of splitting work was less efficient. This was especially so for certain threads where the remainder after dividing the image height by the number of threads (namely threads 9, 12 and 14, where the last worker each took an extra 8 rows compared to the rest of the workers).

2.4 The Modulus Operator

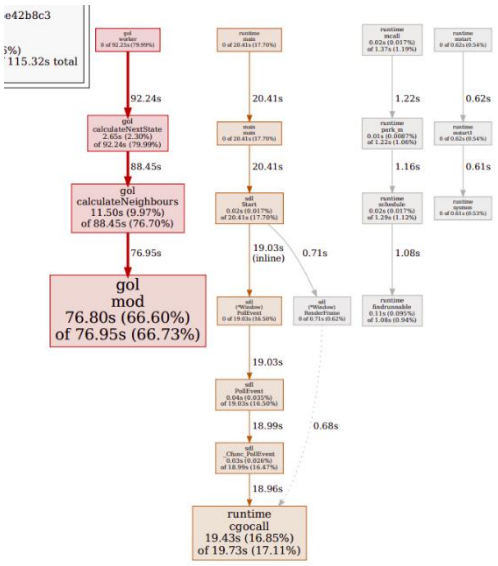


Figure 3: CPU profile of parallel with modules operator

After producing a CPU profile of our parallel implementation, we noticed that a sizable amount of time was spent computing the *mod* function. Our *mod* function made use of Go's built-in modulus operator. We discovered that this is a CPU intensive operation. As a result, we sought to avoid the use of the modulus Operator, especially as it is called every turn for every cell in the world. We resolved this issue by replacing the operator with simple conditional checks.

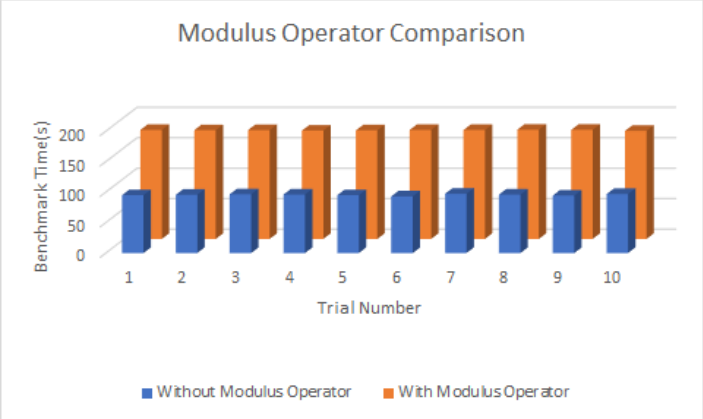


Figure 4: Comparison of benchmark times over 10 trials

Evidently, without the use of the modulus operator, our execution times improved significantly, with the average benchmark time decreasing by a factor of 1.87.

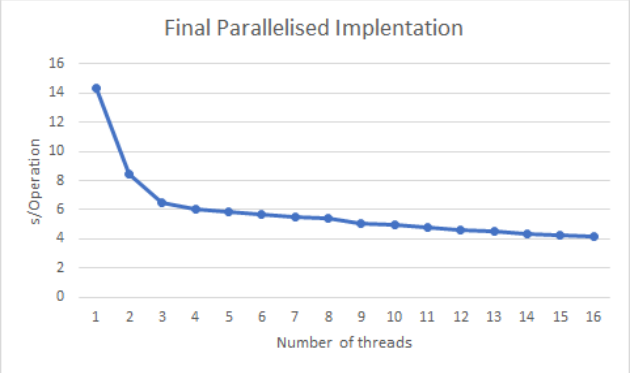


Figure 5: time per operation as thread count increases (512x512x1000)

Figure 5 shows that our parallel implementation - exhibited diminishing marginal returns. This was associated with the increase in channel communication as thread count increases. Another detail we noticed was that as the number of threads increased beyond 12, any increase in performance would be due to external factors. We realise that this is because the lab machines only have 12 threads, so work cannot be split any further.

2.5 Potential Improvements

We observed that our program would have to divide and reconstruct the worlds between the worker goroutines for each turn. This could cause problems in terms of scalability. In order to improve the performance of our parallel implementation, we could implement a halo exchange system, where only the edges are communicated between the workers. The single-threaded distributor function would no longer be reassembling the entire board after each turn, decreasing the amount of data needed to be sent between the goroutines and therefore decreasing runtime.

3. Conclusion

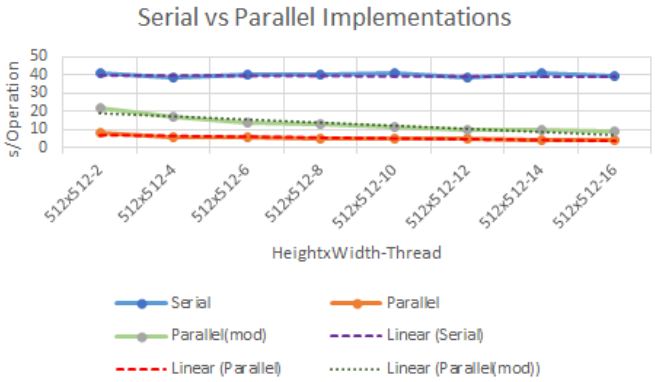


Figure 6: comparison between different implementations for 1000 turns

The performance time has increased significantly over the course of our different implementations, from our initial serial implementation to our final parallel version (without modulus operator).

This assignment allowed us to understand how implementing the principles of concurrent computation can affect a program's efficiency. We observed how the computation speed of a parallelised program can improve by up to 3.9 times when work is distributed amongst worker goroutines efficiently compared to its serial version.

Stage 2 - Distributed Implementation

1. Functionality and Design

1.1 Functionality Implemented

The first part of our distributed implementation involved taking our serial implementation from stage 1 and separating the Game of Life logic that computes the next board state from the SDL controller. Our implementation is based on the publish-subscribe model. This involved setting up RPC connections between the two as the controller would need to send the initial world to the GoL engine and receive the final world from it. For this we created a separate "stubs.go" file that defines the available methods in the GoL engine which at this stage was just *CalculateNextState()*.

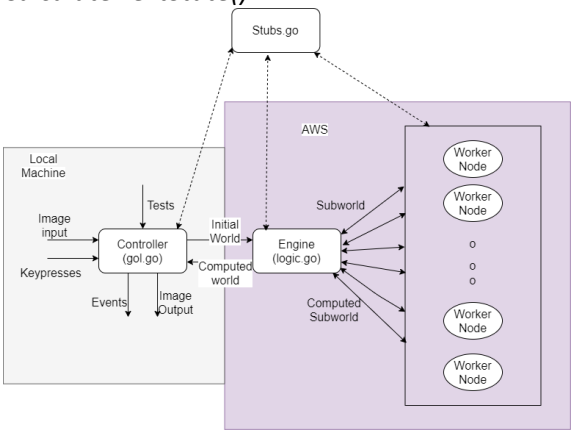


Figure 1: Diagram of System Design

After we had implemented the distributed version of GoL, we added a ticker to return the number of alive cells every 2 seconds and some keypresses. Similar to our parallel implementation, the specific key that the user pressed would be sent down the *keyPresses* channel. For each keypress, we set up another *client.call()* to use the *CallDoKeypresses* method available in the GoL engine. This function would return the current turn and current world to the controller which will then use this information accordingly. The keys we implemented were: 's' to output the current state of the board as a PGM file; 'q' to close the controller only with the engine continuing to compute; 'p' to pause and resume GoL and 'k' to cleanly shut down all components.

Once the keypresses were implemented, the next step was to split up the computation of the GoL board state across multiple worker machines and gather the results together in one place. Our implementation involved a program called "logic.go" that received the initial world from the controller and then distributed the work to the number of worker machines connected to it. We used the same method to split up the work as in parallel, however, this time we used RPC connections to communicate, rather than channels.

1.2 Problems Solved

One issue we came across when implementing the keypresses was that *client.call()* blocked until the procedure in logic returned. This was a problem as we wanted the user to be able to press the keys during runtime. In order to resolve this, we used multiple call functions. We first sent the initial world to logic in order to start the distributor function, returning without waiting for a response. We also used a goroutine to implement our ticker and keypresses to ensure that it would be running during the calls to "logic.go", with the Finally we called logic again but this time to only receive the final world after all turns have been completed.

When connecting multiple workers to logic, we needed a way for logic to know exactly how many workers are available and to distribute work to each one. To solve this, each worker sends its address to logic and logic stores this information in a global array. By doing so, logic only has to iterate through the list of addresses and call each worker.

2. Testing and Critical Analysis

2.1 Distributed implementation with workers

To conduct our benchmarks, we set up multiple c4.xLarge instances on AWS. One instance would run our "logic.go" and the rest would run the workers. We also used x2goclient to access the university's 6-core lab machines to run our controller and benchmarks. Similar to parallel, we completed all of the benchmark testing in one session, with multiple trials per test.

Benchmarking our first distributed implementation yielded these results:

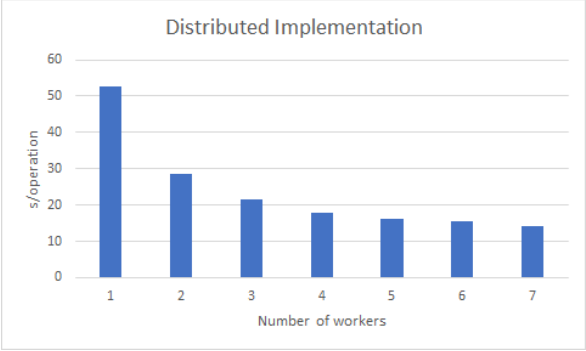


Figure 2: Distributed Parallel Implementation for 512x512x1000

The benchmarks yielded much slower times compared to our parallel implementation. However, that was to be expected due to the slower communications through RPC connections. Similar to our parallel implementation, our distributed one exhibited diminishing marginal returns - as the number of workers increased, the improvement in runtime increased. This suggests that with a much larger number of workers, performance would start to flatten out.
As mentioned in our parallel section, we determined that our modulus function (which is called by *calculateNeighbours*) was very CPU intensive, and this was the same case for our distributed implementation.

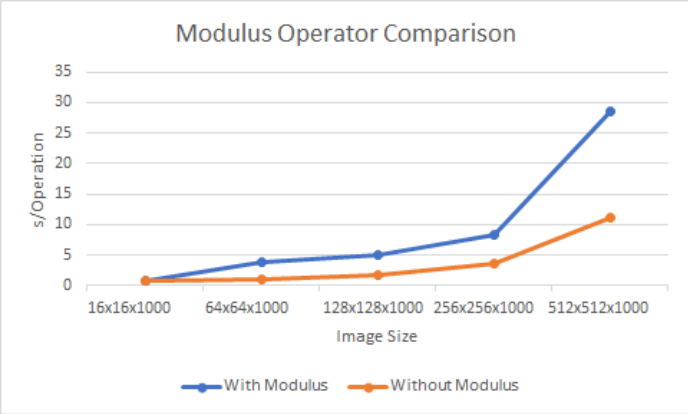


Figure 3: Comparing benchmark times for distributed with and without MOD, 2 workers on all images

If we had kept the modulus function, this would have reduced scalability as a large proportion of the runtime would be spent computing that function. Figure 3 shows that the overall benchmarking for the version without the modulus function is 2.3 times faster than the version with the function.

2.2 Potential Improvements

To improve the efficiency of communication between worker nodes, we could have implemented a halo exchange scheme where only the edge rows would need to be communicated between the nodes for each turn. This would ensure that the game logic for each section is worked internally by each worker and avoids sending huge amounts of data every turn. By implementing this scheme, the scalability and performance of the program would have been vastly improved.

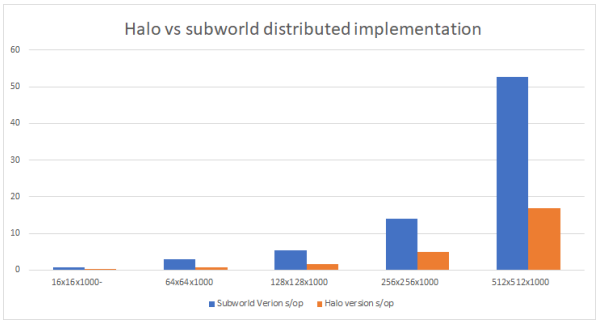


Figure 4: Halo vs Sub-world implementation for 1 worker

We were able to successfully implement a halo exchange implementation for 1 worker. As evident in figure 4, the halo implementation is much more efficient in comparison with a version that does not implement the halo exchange scheme. We would expect that as the number of workers increase, this improvement in scalability would be further emphasised.

Another possible improvement would be to implement a parallel distributed system with multiple worker threads per machine. Each worker would take the section it receives and split it up further so that it can calculate each subsection concurrently. This along with the halo exchange scheme would make our distributed implementation much more efficient.

Our implementation currently has no solutions for dealing with components of the system disappearing due to reasons such as broken network connections. To rectify this, we could have our logic check to see if the TCP connection is closed before calling each worker by reading from the connection and checking to see if it returns 0 bytes. This ensures that if the connection is broken, our logic does not attempt to send to a closed connection and produce an error.

Currently, our distributed implementation has no visualisation for the game state. This could be another improvement. Previously, we had successfully implemented the SDL visualisation, but it failed to pass all the tests (due to sending on closed channels).

3. Conclusion

From the distributed side of the assignment, we have seen how distributing a serial program amongst multiple workers can positively affect the overall computation speed by around 2.2 times. However, there are also many ways to further improve our implementation.