Exploring Parallel and Distributed Implementations of Conway's Game of Life

30th November 2023

Computer Systems A Coursework

Matthew Pidden
bb22475
BSc Computer Science
University of Bristol

Matthew Cudby
oh22896
BSc Computer Science
University of Bristol

***Abstract*—This report delves into the exploration of various computational implementations of Conway's Game of Life, encompassing a serial approach, multiple parallel processing strategies, distributed computing models, and a synthesis of parallel and distributed techniques, providing a comprehensive analysis of their performance, scalability, computational efficiency, memory and network usage.**

I. Game of Life Simulation

Conway's Game of Life is a cellular automaton simulation devised by mathematician John Conway. It operates on a grid of cells, where each cell evolves through successive iterations based on simple rules, creating intricate and often unpredictable patterns.

II. Parallel Implementation

Parallel processing is a computing paradigm where multiple tasks or processes are executed simultaneously, improving overall system performance by leveraging the capabilities of multi-core processors.

A. *Functionality and Design*

*1) **Functionality Implemented**:* Initially when first creating a correct execution of Conway's simulation, we did not utilise any concurrent or distributed methodologies. This serial implementation ran the cellular automaton one turn at a time, operating on each cell individually and sequentially. Naturally, this implementation was not particularly fast, as the majority of the CPU's threads were not in use [Fig 2]. Hence we worked on creating a parallel solution, to efficiently utilize available core threads. Our approach to this was to split the the grid of cells into sections of multiple rows, and operate on multiple sections simultaneously. This is made possible by modern computer's multi-threaded processors.

As we implemented this project using *Google's Go Language*, our initial parallel implementation was based of the use of *Channels* which are a communication mechanism that facilitates communication and synchronization between concurrent processes through message passing. We also used *Go Routines* to execute the concurrent execution. On each iteration of the process, the grid is sent to multiple *Worker Threads*, along with the upper and lower bound row indexes. This ensures each *Worker Thread* has access to the neighbouring cells of it's assigned section. Once each section has been processed, the *Worker Thread* returns only it's assigned rows back via a *Channel*. In the main *Distributor* function, a for loop is awaiting each sections return in their respective channels.

To ensure correctness, we added a live reporting of the number of alive cells every 2 seconds, along with the output of a pgm formatted image upon completion. To enhance usability, certain key presses were added, allowing a user to pause, halt, or output an image of the current state. These features all utilise *Channels* to an input output management system.

*2) **Problems Solved**:* The first problem we encountered when implementing a parallel version was due to the closed domain property of Conway's simulation, effectively creating an infinite grid of cells. The *Worker Threads'* inability to access cells that were on an opposite edge or corner to the current cell meant we had to pass the entire cell grid to each worker. This is not a memory efficient solution [Fig 7] and hence we implemented a memory sharing model with a significant improvement in memory usage [Fig 7].

Secondly, we initially split the grid into sections of rows for the *Worker Threads*, a simple floor division was used to calculate the sections height.

$$height_{section} = height_{image} // threads$$

however this left remainder rows which had to be included in the final section. This did not provide an even distribution of rows between section, therefore to solve this problem we implemented a dynamic row distribution algorithm [B.3].

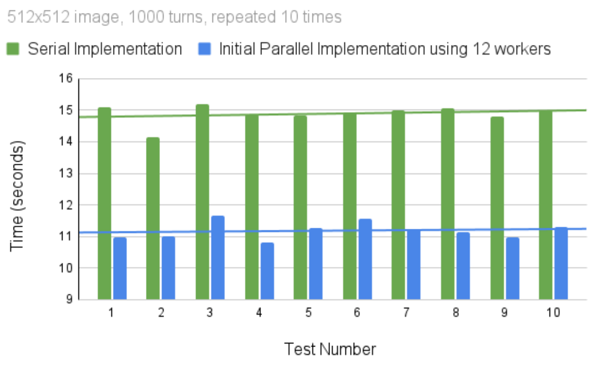

Fig. 1. Serial implementation graphed against a basic parallel implementation for a 512x512 image run for 1000 iterations, repeated 10 times.

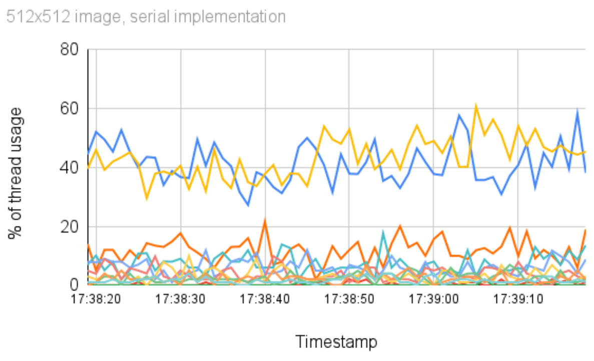

Fig. 2. CPU thread usage as a percentage throughout a one minute interval of running the serial implementation on an Intel(R) i7-1255U CPU with 12 threads.

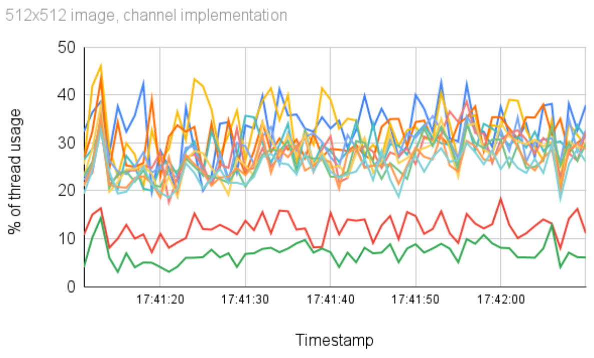

Fig. 3. CPU thread usage as a percentage throughout a one minute interval of running the channel based parallel implementation on an Intel(R) i7-1255U CPU with 12 threads.

B. *Testing and Critical Analysis*

1) ***Acquiring Results**:* To acquire data for analysis, we used *Go's* built in *Benchmarking* on a personal laptop with a Intel(R) i7-1255U CPU with 12 threads. We chose parameters that would allow us to compare the various implementations and their expected performance difference. The raw data was then converted in a universal csv file and uploaded to *Google Sheets* for visualisation.

The exception to that method is when acquiring data on CPU thread usage [Fig 2][Fig 3]. We used *'sysstat'*, a performance monitoring tool for Linux, with the *'sar'* command to obtain raw CPU utilization data. Then a custom *Python* program to convert that data in a usable csv format for analysis and visualisation. Likewise to obtain the CPU profile [Fig 6], we used *'pprof'*, a performance profiling tool for *Go* programs.

2) ***Serial vs Basic Parallel Implementation**:* Our initial parallel implementation clearly obtains a faster average run time than our serial version [Fig 1]. We calculated the percentage decrease in run time,

$$\left(\frac{14.89 - 11.18}{14.89}\right) \times 100 = -24.91\%$$

using an average of 10 repeated tests for each implementation. Our parallel implementation scaled well with an increased number of worker threads [Fig 5]. As a further indication of successful concurrency, all threads of the CPU were undergoing significant utilization whilst the simulation was run [Fig 3]. As seen, the initial increase in performance is significant however performance improvements taper of beyond 12 worker threads [Fig 5]. This is likely due to the physical limitations of the thread count used for these benchmarks [B.1] and given an increased thread count, our implementation's performance would likely continue to scale beyond 12 worker threads *Worker Threads*.

3) ***Optimisation of Thread Split**:* By incorporating a dynamic row distribution algorithm, in our case a basic Round Robin scheduling method, we aimed to optimize the distribution of work among threads. By evenly distributing the rows to be processed between the workers, we minimize the chance of a single worker having significantly more work than others. Previously in the worst-case scenario, with a 512x512 image, the maximum number of extra rows a thread may have to compute by itself was 8. This was repeated every single iteration however with the row distribution optimization the worst case scenario was a worker having to process just a single additional row. Whilst this approach did enhance the overall run time efficiency [Fig 4] of our paralleled implementation, the performance improvements were minor.

4) ***Memory Sharing Implementation**:* Memory sharing is a concurrent technique where multiple *Go Routines* access shared data directly in memory, allowing them to communicate and synchronize without the use of channels. In theory this can provide improved run times, as there is no longer the overhead of message passing and synchronization mechanisms associated with channels. In

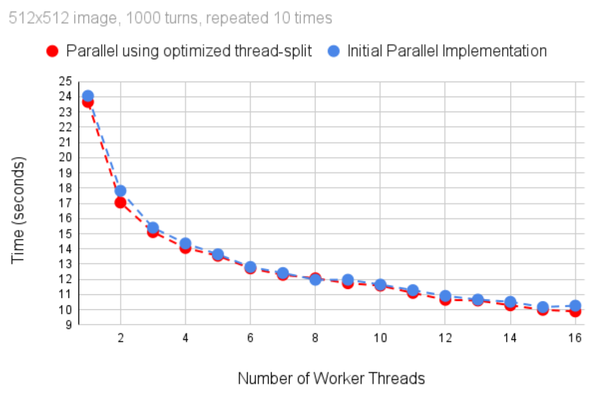

Fig. 4. Non-optimized thread-split graphed against an optimised thread split for a 512x512 image run for 1000 iterations, repeated 10 times, using 1 through 16 worker threads.

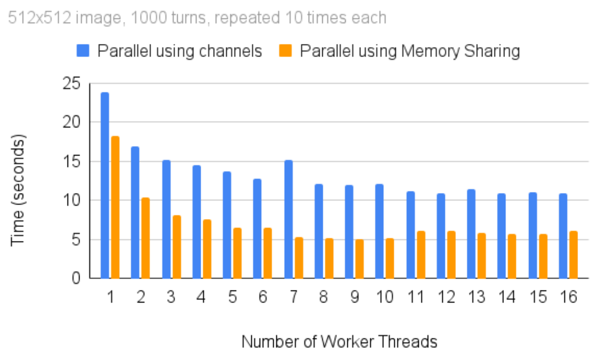

Fig. 5. Memory sharing parallel implementation graphed against a parallel implementation using channels for a 512x512 image run for 1000 iterations, repeated 10 times.

our memory sharing implementation, two shared grids are used. The first provides the data *Go Routine* required to process a turn of GoL, and the second is for the updated cell states of each iteration. Our memory sharing implementation has an average percentage decrease in run time of 46% against the channel based implementation [Fig 5]. Furthermore the memory shared model decreased run time from our initial serial implementation by 65.8%. When we analysed the CPU profile of our channel based approach, one noticeable cause for concern was the high number of calls to a closure function used to store the matrix in an immutable state, thus removing the possibility of a developer introduced race condition. However given that calling this function effectively replaces the functionally of reading data from an array, the high number of calls to the function is understandable. The performance implications of using this closure function, instead of a regular array is something that requires further exploration.

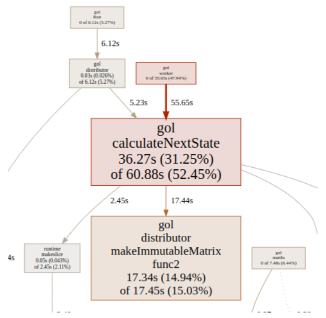

Fig. 6. A CPU profile for the parallel implementation using channels.

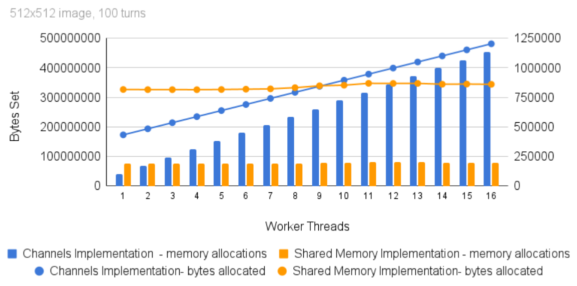

Fig. 7. Memory allocations and bytes for a memory sharing parallel implementation against a channel orientated parallel implementation for a 512x512 image run for 1000 iterations

C. Potential Improvements

1) Memory and Byte Allocation: Whilst we did manage to decrease run time by up to 65% from our initial serial implementation, one important consideration in the efficiency of our implementations is the frequency of memory allocations and the size in bytes of these allocations. This is a crucial factor to consider as it reduces resource usage, improves performance, lowers latency and enhances scale-ability. The memory sharing implementation is superior to the channel based implementation [Fig 7], with regards to the bytes allocated metric, when the number of worker threads is above 9, and inferior when below. With regard to memory allocations, the shared memory implementation has significantly few allocations in cases with three or more worker threads [Fig 7]. Note that whilst the shared memory model bytes allocated data appears constant in this graph, in reality bytes allocated scales linearly with image size.

D. Parallel Conclusion

In conclusion, throughout our experimentation with parallel implementations, we managed to achieve a significant decrease in the run time of Conway's simulation with the most significant performance increases being made by the memory sharing implementation. Both of our parallel models scaled well with increased worker threads which was encouraging, and highlights the benefits of concurrent programming. In future work, it would be interesting to explore methods for decreasing memory usage and allocations given that it is on these metrics that our implementations fall short.

III. Distributed Implementation

Distributed programming is a computing paradigm where components, located on different networked computers, communicate in order to achieve a common goal. Our distributed implementations aimed to improve the rate at which Game of Life simulations could be completed by splitting the workload of the calculations between multiple systems. By utilizing the distributed approach we aimed to have our implementations be able to handle larger workloads, function with enhanced fault tolerance, and to have their performance scale with the usage of additional computational resources; such that their performance could surpass that achievable on a single machine or processor.

A. *Functionality and Design*

1) ***Functionality Implemented:*** When distributing a program using *Google's Go Language*, *Remote Procedure Calls (RPCs)* are used as a method to call functions on nodes of the network. When considering a distributed model for the Game of Life simulation, the simplest implementation uses a single local controller to call a *RPC* on a Game of Life (GoL) engine. These GoL engines, are comparable to our parallel worker threads, whereby the core simulation logic is executed. Upon completion of process, the engine returns the final state of the world back to the local controller. Whilst this implementation is a valid distributed model, it does not utilize the full ability of the distributed paradigm, as only one engine is used.

To upgrade our system, we increased to four engines, along with a broker. The purpose of the broker is to reduce coupling between the local controller and the set of engines. This allows the local controller to call a single function on the broker, which in turn communicates with the engine set. In this implementation the broker splits the grid into sections of rows, just like our parallel models, and calls an *RPC* function to send each section of rows, out to the engines. Upon completion of processing on each engine, the section is returned to the broker, who stitches the grid back together, and iterates. To ensure correctness, we again added a reporting of the alive cell count from the broker to the local controller every 2 seconds. This was achieved via a separate *RPC* call to the broker which queries the state of the world. The local controller then counts the alive cells, and displays a live snapshot of the simulation to the user. We chose the only display a snapshot every 2 seconds to minimise network traffic, which otherwise can be a bottleneck and impact efficiency. To improve usability, we enabled key press

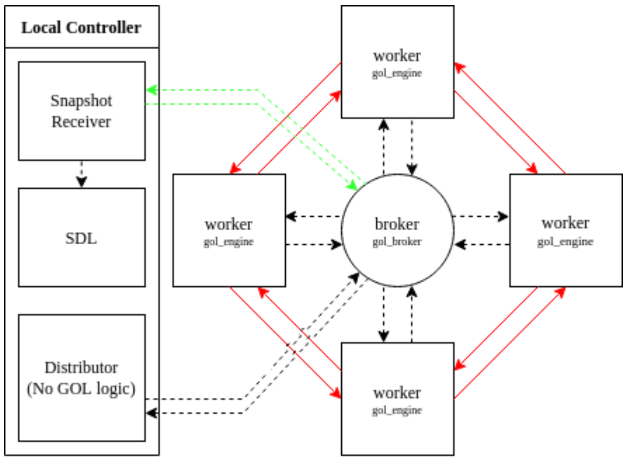

Fig. 8. Diagram of Halo Exchange implementation with 4 workers

functionality allowing the user to pause, output a snapshot, quit the local controller, or kill all the distributed components.

One important property of the distributed paradigm is fault tolerance, which helps systems continue to operate smoothly, or start back up, even when an individual component fails or is shut down. In our system we use a boolean flag, which when set to true allows a local controller to take back over the processing of a previously disconnected or shut down controller.

Whilst our distributed system was robust and efficient at this stage, we endeavoured to create even more efficient models. Our first attempt was combining our parallel logic onto our distributed engines. This means that for each section an engine received, it split the section into new sub sections and concurrently executed on those. This requires no extra *RPC* calls, so the network traffic remains constant, however the traffic is quite high already, as the broker sends every engine their section on each iteration. When considered, each engine only needs to know a small proportion of the other engine's result. The middle of their respective sections has no need to be sent across the network.

From these ideas we implemented the Halo Exchange [Fig 11]. The exchange implementation works by having each of the workers create a client-server connection with the two workers that contain the matrix sections adjacent to its own. Before a step of the simulation is completed, each worker makes an RPC call (both occurring concurrently within separate go routines) to both it's neighbours requesting the row above and below its own section; These rows are required for the completion of a GoL step on its own section. In turn each worker is concurrently running a piece of code to serve these incoming GetRowRequests. Once both 'halo-rows' have been received and both the top and bottom row of its own section have been sent of, a step of GoL is calculated. Whilst this step is being calculated any incoming GetRowRequests from workers that have completed their own step calculations first are blocked until the held section matrix is updated with the results of the GoL calculations. This steps continue until the target number of GoL steps have been simulated. The complete section are then sent back to the broker, pieced together into a complete GoL matrix, and then passed back to the local controller. When displaying a snapshot to the user, the broker has to make an additional request to each of the workers. The workers send back the current state of their section. The broker then pieces them together and sends the grid back to the local controller to be displayed via SDL.

*2) **Problems Solved**:* When implementing the distributed implementations one problem we struggled with was managing the *RPC* calls to the broker on the local controller. This was because calling an *RPC* call is a blocking statement, meaning the program will wait at that line of execution until the *RPC* returns. The local controller calls multiple *RPC* calls throughout execution besides the main broker method, including *RPC* calls for the most recent GoL grid, and to update the components of any state changes. To ensure the *RPC* calls could be made simultaneously, *Go Routines* were utilized with *Channels*, which ensured exiting of the *Go Routines*.

B. Testing and Critical Analysis

*1) **Acquiring Results**:* Similar to our parallel bench marking, we used *Go's* built in *Benchmarking* tool. To run our distributed nodes, we used *Amazon Web Services (AWS) m5.large* instances. We again chose parameters that

TABLE I

Host	Name	vCPU	Threads / core	Memory
AWS	m5.large	2	2	8 (GiB)

would allow us to compare the various implementations and their expected performance difference. *Google Sheets* was again used for visualisation of our data.

One consideration we had to take into account for our distributed analysis is the overhead of establishing connections over the network, and continuous delay of communicating over a network. To reduce the variance caused by the networks, we ran our simulations for an increased number of iterations. Consequently, we anticipate that our results effectively highlight distinctions between various models and computationally diverse implementations.

*2) **Initial Distributed Implementations**:* Our initial distributed implementation was expected to be slow, due to only utilizing a singular engine. Upon increasing to four engines with a broker, run time was decreased by 66.75% on average over five repeated tests [Fig 8]. This is because the computation of the cells, is split onto four

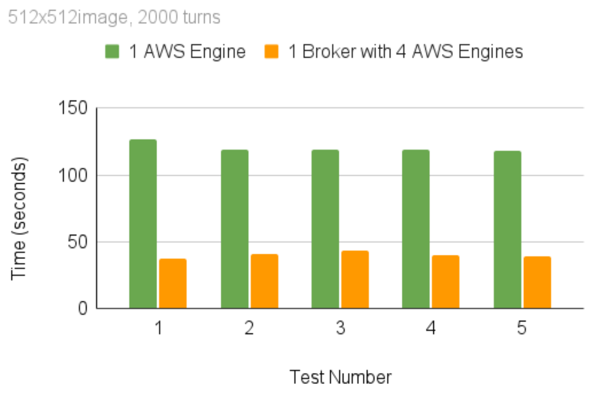

Fig. 9. Distributed implementation with one AWS engine, graphed against a broker and 4 AWS engines implementation on a 512x512 image for 2000 iterations.

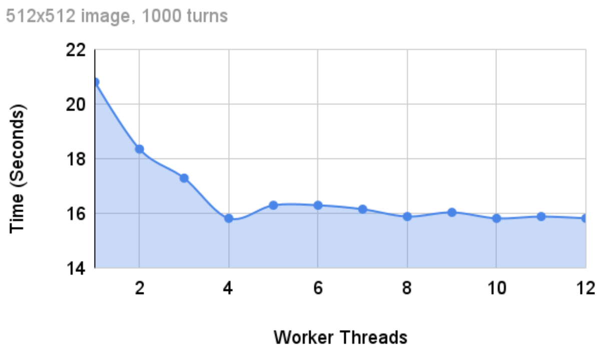

Fig. 10. Execution time for a parallel distributed implementation with varying number of workers on each engine node for a 512x512 image run for 1000 iterations.

separate instances, each with their own physical CPU to process the data.

*3) **Paralleling Engines**:* Whilst our paralleled engines are more efficient than our serial engines, the decrease in run time was not as great as expected [Fig 10]. This is likely due to the fact that an *m5.large* AWS instance, contains two *vCPUs*, which each posses two threads [Table 1]. From this we concluded that the paralleled engines only scaled well up to four worker threads and our bench marking confirmed this [Fig 9]. Another reason for the disappointing results could be that the added computation of splitting the section into subsections outweighs the benefit's of the concurrent execution. In fact the paralleled engines only delivered decreased run times when the image was of size 512x512 [Fig 12]. For smaller image sizes the run time was either equal or increased compared to the equivalent serial model [Fig 12]. In conclusion, the paralleled engines do have a decreased run time, however, only when run on a large enough image, with a sufficiently large iterations set [Fig 10].

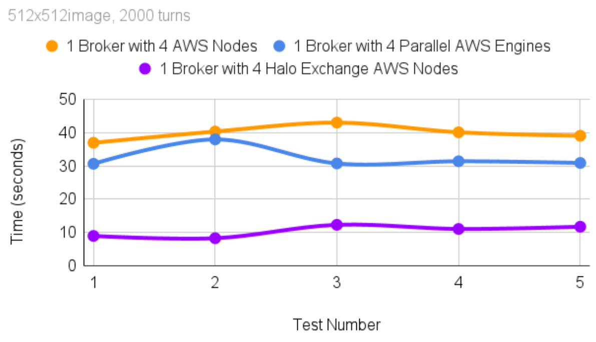

Fig. 11. Multiple 4 AWS engine implementation graphed against each other on a 512x512 image for 2000 iterations.

*4) **Halo Exchange Method**:* The Halo Exchange implementation boasts significantly decreased run time in comparison to to our other four node distributed implementations [Fig 9]. When run on a 512x512 image for 200 iterations, Halo Exchange decreased run time by 67% in comparison to our parallel distributed model, and 73% decrease from our serial four broker model. This is most likely due to the workers no long having to receive requests from the broker for every step of the simulation due to them instead communicating only the necessary data between themselves.

The Halo Exchange implementation run time only gradually increased with increase in image size [Fig 12]. The results show that the Halo Exchange implementation is the best choice when simulating GoL on image sizes above 256x256 and a poor choice for images smaller than that. The relatively poor performance for small image sizes is likely due the the significant overhead of inter-worker communication (GetRowRequests), the frequency of which is not effected by image size.

The only slight increase in run time with respect to the increase in image size suggests that the majority of the run time is a result of workers making requests between each other and not of calculations being made by each worker.

C. Potential Improvements

*1) **Reduce size of packets**:* Given that each cell in the GoL matrix can be in one of two states (alive or dead) we could streamline the representation of the matrix such that each cell is represented by a bit.

$$bit = \begin{cases} 0 & \text{if cell is dead,} \\ 1 & \text{if cell is alive} \end{cases}$$

By using bits instead of the uint8 data type to represent the state of the cells you could reduce the size of the GoL matrix by a factor of 8 thus reducing the time to transfer information about the state of the matrix between components of our distributed system.

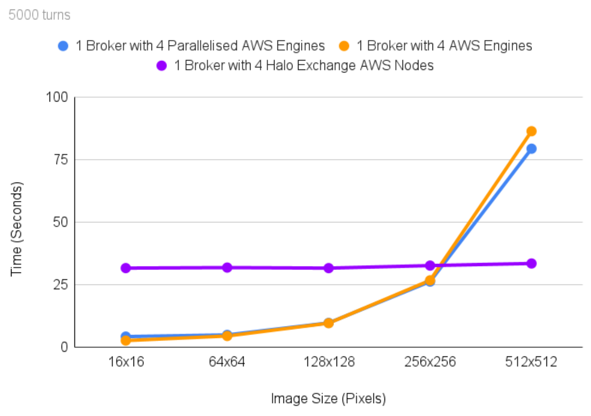

Fig. 12. Multiple 4 AWS engine implementation graphed against each other for 5000 iterations, over multiple image sizes.

D. Distributed Conclusion

Through our development of several distributed solutions we have managed to significantly improve the rate of GoL simulations in comparison to the base serial version. In particular we found that Halo Exchange was by far the quickest when it came to processing large images sizes, and that it's run time scaled the slowest with respect to the image size. Therefore if further development were to continue it should be based of the core design of this implementation

IV. Final Conclusion

In conclusion, we have provided a comprehensive exploration of both parallel and distributed implementations of Conway's Game of life. The data we have gathered offers valuable insights into the performance of each of these solutions under varying inputs and circumstances. The bench marking and testing of a diverse range of scenarios has provided the necessary information to inform a decision upon the optimum implementation given a specified input size and known hardware constraints. Whilst our work has yielded valuable insights, there remains room for further refinement and exploration of methods for improving efficiency and resource usage, and the enhancement of fault tolerance mechanisms.

References

[1] Amazon Web Services. *Amazon EC2 Instance Types*. https://aws.amazon.com/ec2/instance-types/. 2023.
[2] Go Devs. *Testing and Benchmarking*. https://pkg.go.dev/testing/. 2023.
[3] Stanford University. *Game of Life*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/2001-02/cellular-automata/index.html. 2023.
[4] Wikipedia. *Round Robin Scheduling*. https://en.wikipedia.org/wiki/Round-robin_scheduling. 2023.