COMS20008 Computer System A
Parallel and Distributed Implementation of Game of Life (Gol)

Gordon Wai Hin Kam, li23179 Ka Ho Leung, nm22031 Sze Chai Yuen, ee23871

1 Introduction

This report presents an analysis of the implementation of parallel and distributed system, focusing on the benchmarks, scalability, and system resilience within networked environments. We examined goroutines used for parallelism, network communication and system design in a distributed system, presenting results obtained through empirical measurement and formal analysis. Additionally, we explored three extensions for the distributed implementation in this project.

2 Parallel Implementation

2.1 Goroutines

2.1.1 Key Press Event

Firstly, the `ManageKeyPress(..)` goroutine handles key press events from the distributor's channel when a user presses a key. Then, it sends a signal corresponding to the event to the `type KeyPressChannel struct`, notifying the distributor of the required operation for the next turn within a **for-select** loop. In this case, we can avoid **race condition** for instantaneous key pressing during the evolution of Gol because we can ensure that each key press signals are received by the distributor in the **select** statement.

2.1.2 Ticker

Next, `ReportAliveCells(..)` creates a ticker from time[1] package. Here, a **for-select** loop handles the signals from the ticker every 2 seconds, as well as a quit signal to stop the ticker once all turns in the Gol are completed. A **Mutex Lock** is also used to prevent race conditions during asynchronous **read/write** operations on the global variable `GameState`. This allows each worker to interact with the `GameState` safely while sending `AliveCellsCount` to the event channel.

2.1.3 Multi-Threading Implementation of Gol Logic

After that, the `DelegateStateWork(..)` and `DelegateCellWork(..)` functions created a `WorkerChannels` containing two slices of channels, namely `StateWorkerChannels` and `CellWorkerChannels`. These slices of channels are used to gather and organise each worker's results. Each result is subsequently iterated over, by appending and passing them to the export channel.
Additionally, instead of passing the world with `type [][] byte`, each worker goroutine receives an immutable world function, namely `type func(int, int) byte`. This **closure** method ensures that no workers can directly access the world and modify it, improving safety and preventing race condition.
While a **Message-Passing** model with channels was chosen to delegate work for workers, we could have used a

[1]Time, Go Package, https://pkg.go.dev/time

Memory-Sharing model as an alternative. In particular, a worker would acquire a **Mutex Lock** to access the **critical section** (e.g., world), while other workers are waiting for that worker's lock to release. However, this implementation would introduce risks that programmers may forget to manually release the lock, which can result in a **deadlock**. Considering the factors of readability and complexity, we ultimately chose the **Message-Passing** model to implement multi-threading Gol Logic.

2.2 Benchmark Results and Analysis

2.2.1 Benchmark Results

To evaluate the concurrency performance, we conducted Go benchmarks by varying the number of worker threads and repeated the test 6 times to calculate the average runtime of each number of threads (in seconds). Each test used an input size of 512×512 with 1000 turns. All measurements were collected on Gordon Wai Hin Kam's Apple MacBook Pro M1 chip 8-Core CPU.

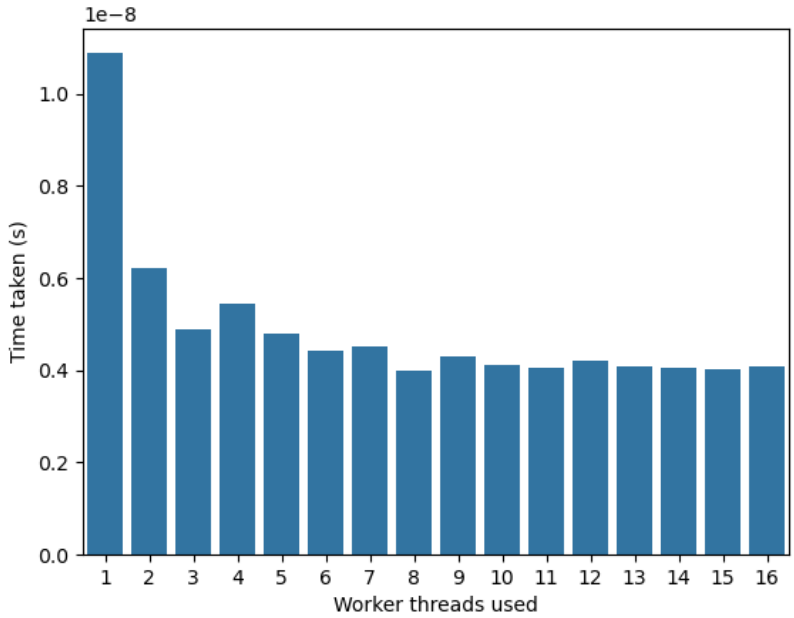

Figure 1: Benchmark Results of Input 512×512×1000 varying number of threads used

2.2.2 Analysis

As the number of worker threads increases, runtime decreases exponentially. Using 16 worker threads is **2.75**× faster than using a serialised single-threaded worker implementation (shown in Figure 1). In the single-thread implementation, operations often wait for I/O or other resources, creating a **bottleneck** by idling the CPU. In contrast, multi-threading implementation allows other threads to continue their executions while one thread is waiting for resources.

The improvement of performance from 8 threads to 16 threads, compared to from 1 thread to 2 threads started to diminish. Ideally, we would expect that the relation of worker threads used will be inversely proportional to the runtime (i.e. if we double the number of threads used, we will halve the runtime). However, the rate of performance improvement begins to decrease due to hardware limitations. Since there are only 8 physical cores in the Apple M1 CPU, exceeding the threshold of physical cores in hardware, it will not benefit much on the performance. Moreover, managing a large number of threads can introduce overhead, leading to frequent context-switching by the CPU, which will reduce the performance. As an example (shown in Figure 1), while we would

expect 12 worker threads to perform faster than 8 worker threads, 8 worker threads' runtime $\approx 4.1\,\text{s} < 4.5\,\text{s} \approx$ 12 worker threads' runtime in 1000 turns.

Furthermore, unlike Intel CPUs, the Apple M1 chip does not have hyper-threading[2]. Instead, it uses 4 high-performance cores (Firestorm) and 4 high-efficiency cores (Icestorm) to optimise performance. This might be the reason why Intel can improve slightly better than Apple while exceeding the number of physical CPU Core.

3 Distributed Part

3.1 System Design and Design Choices

Starting with the design, the distributed system consists of three components: **controller**, **broker** and a list of **Gol workers** (shown in Figure 2), and all shared **Request** and **Response** types are defined in `stubs.go` file. The reason that we chose to design this system architecture was that introducing a broker can decouple the controller and workers, even if one of them malfunctioned, the whole system will not shut down. Moreover, it also benefits **Encapsulation** of Gol logic in the server where the client cannot directly invoke those methods. Additionally, we have utilised parallelisation on each worker to create a **Parallel-Distributed System**.

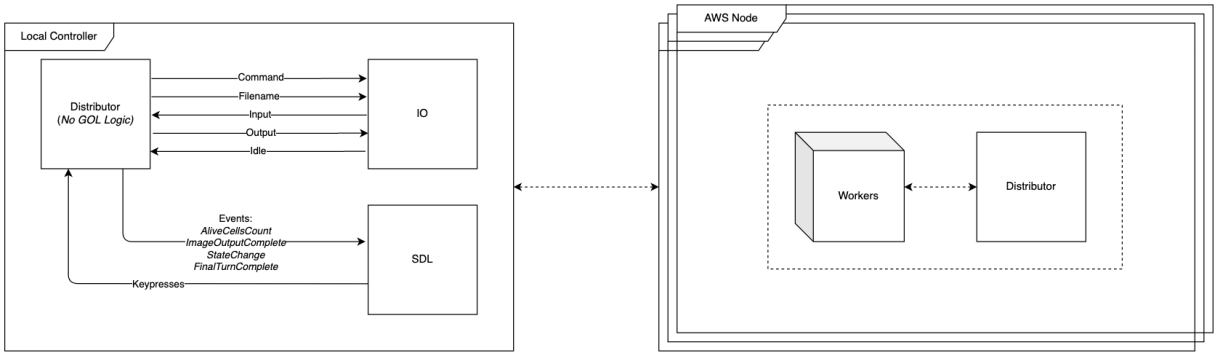

Figure 2: Diagram of Distributed System Architecture

3.2 Network Communication

To instantiate the system, the broker needs to be started as a **bridge** for the controller and Gol workers. Then, workers can start registering their connection using **Remote Procedure Call** `Broker.Register` with the broker. Finally, the **client** can start running the game by calling the broker to start processing using `Broker.RunGol`, thus the broker will start interacting with the workers up to the `GameState` of the last turn, and it will send the resulting world back to the client every turn until the last turn.

As we talked briefly about the communication in the system in section 3.1, we can see that the client does not directly invoke any server methods that ask for evolutions. Instead, the client uses **Remote Procedure Call (RPC)** to interact with a remote component (e.g., the Broker), allowing it to extend its functionality by offloading tasks to other parts of the system. The benefits of RPC enable different components of a system to be written in different programming languages as well as hosted on various platforms (e.g. using AWS instances) which increases flexibility and diversity regarding network infrastructure.

3.3 Functionality of Components

Initially, the client started **goroutines** for listening to keypress events and reporting **AliveCells** every two seconds by using an RPC to broker. For the keypress events, when the client presses a key, it will then send the

[2]Intel Hyper-Threading Technology is a hardware innovation that allows more than one thread to run on each core.

event to the broker, and the broker will handle the event, telling all workers to do tasks that the client requires. For example, when the client asks for **exporting world**, the broker will tell all workers to send their current `sliceWorld` back to itself, and the broker will ultimately send the combined world back to the client.

When the broker receives the RPC `Broker.RunGol` from the client, it will first **initialise** the game by **splitting** the world into slices and sending the `sliceWorld` to those workers using `Worker.Initialise` (**note**: `w` for `Worker struct`). After initialising, the broker will then start a **for-loop**, looping through **the turn number**. In each iteration, the broker will start calling `calculateStateAndCells(..)`, and during this subroutine, it will set up a **list of channels** for receiving the resulting `sliceWorld`, and call `Worker.ProcessTurn` to each worker **concurrently** via `Client.Go`. The broker will then join all `sliceWorld` in the correct order, and send all the resulting information back to the client.

In each worker, when they receive an RPC `w.ProcessTurn` from the broker, it will first call `w.haloExchange(..)` to exchange **halo regions** with the previous and the next worker. After the exchanging is finished, the worker will start a goroutine `DelegateStateWork(..)` and start processing the next gameState in **parallel**. After that, the worker will **filter** the halo regions with `w.filterHaloRegion(..)` and start processing the **AliveCells** by calling goroutine `DelegateCellWork(..)` in **parallel**. Finally, the worker sends the results back to the broker, and the whole process is done for a turn.

3.4 Scalability with Additional Components

For **Halo Exchange** extension, when the worker receives the `sliceWorld` from the broker, it will call `w.haloExchange(..)` first. In this subroutine, the worker invokes `SendBottomRegion()` and `SendTopRegion()` to the other workers if there is more than one worker. After the worker receives the halo regions, it will call `sendReadyToOthers` to other workers to make sure all related workers, including itself, finish the **halo exchanging** process before processing the `sliceWorld` of the next turn. After that, the worker will add the halo regions to its own world, and start processing the world to the next state in **parallel**. Also, it is notable that no halo exchanging will take place when the worker **initialises** the world or **only one worker** is registered in the distributed system for optimisation.

We have used a **Peer to Peer** communication scheme for Halo Exchange instead of using a centralised Broker. Implementing this scheme ensures no communication overhead for synchronisation on the broker. On the other hand, implementing a suitable algorithm for synchronisation on the distributed system could be difficult. We have used **Barrier Synchronisation**[3] scheme where after a worker has received both Halo Region, it will make an RPC call to all the workers to notify them by sending a signal to the buffered `w.SyncChannel` with the size of the number of neighbours. Then, it will end the loop and listen for all signals sent from other workers. This method can avoid workers being at different phases of `ProcessTurn(..)` since there is no global clock to synchronise servers at a particular stage in distributed systems.

For **Parallel-Distributed system** extension, when the worker starts the goroutine `DelegateStateWork(..)` to process the world for the next turn, the `sliceWorld` in itself will be sliced again into `p.Threads` parts by specifying the number `startY` and `endY`. Each slice will then be passed into `p.Threads` goroutines `StateWorker(..)` to process. This makes the worker process the `sliceWorld` in parallel. Also, knowing that the world processing in each worker is already sliced by the broker, and the worker will slice it once again to its **threads**, the runtime of the program might improve even more. Similarly, `DelegateCellWork(..)` used a similar idea to process the list of **AliveCells** but also required an extra offset (`startY` from the broker splitting the original world).

[3]Desai Jay, "Barrier Synchronization in Threads", 2020, https://medium.com/@jaydesai36/barrier-synchronization-in-threads-3c56f947047

3.5 Methodology for Measurements and Analysis

All the distributed implementation measurements were collected on Ka Ho Leung's Rog Zephyrus G15 8-Core CPU in WSL2 by running a Gol Benchmark on a 512×512 PGM file. In particular, we will focus on the performance of the execution of Gol in a distributed system of two graphs, **Halo Exchange on Centralised Broker vs. Direct Exchange between Servers** and **Serialise Distributed System vs. Parallel Distributed System**.

3.5.1 Parallel Distributed Performance

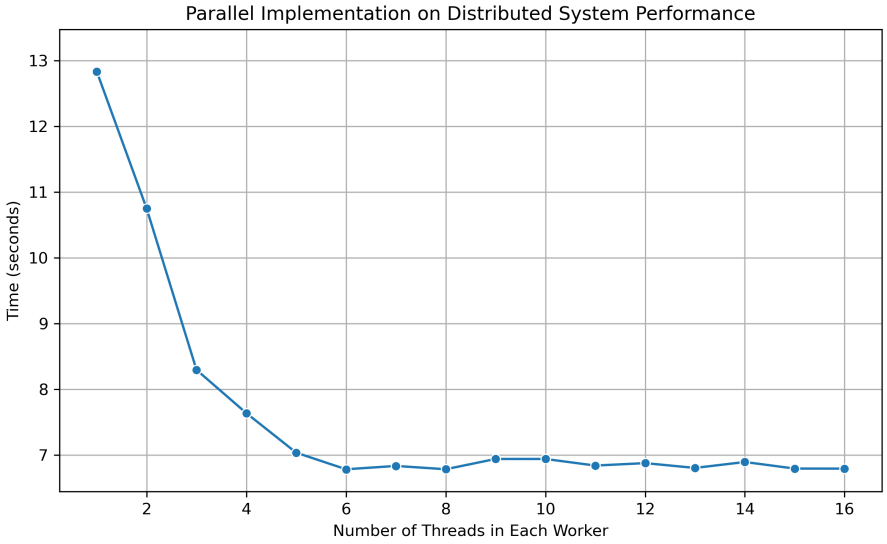

Figure 3: Performance of Parallel System

For the parallel distributed system, we ran our benchmark with fixed 8 **AWS Node** as **Gol Workers**, and varied the **number of threads** in each worker. Similar to parallel implementation in **2.2**, the runtime decreases exponentially as the number of threads in each worker increases. Specifically, the runtime with 8 **16-threaded** workers is about **1.8**x faster than running with 8 **single-threaded** workers (shown in Figure 3).

Interestingly, if we try to compare the result in Figure 1 in section **2.2.1** with Figure 3, we can see that although both of them utilitise parallelization, the performance of Parallel-Distributed system is still far slower than local parallel implementation.

The improvement in the runtime became insignificant after having more than 6 worker threads. This is because, inside the distributed system, the runtime depends more on the communication between servers. Therefore, even though the workers use less time to process `sliceWorld` with multiple threads, they use more time to communicate with their adjacent **AWS Nodes** and the broker. As a result, the runtime improvement is way less than **2.75**x compared to the benchmark of the parallel implementation in Figure 1 in section **2.2.1**.

3.5.2 Halo Exchange

For Halo Exchange implementation, we ran our benchmark by varying 8, the number of **AWS Node**. Our Halo Exchange implementation is based on a Parallel-Distributed System. We notice there is a trend that as the number of workers increases, **Direct Halo Exchange** between servers performs better than **Broker-Server Exchange** (Shown in Figure 4). In Centralised Broker Halo Exchange, every worker has to resync on the Broker

which causes a heavy communication overhead on the Broker node while each server communicates with other neighbours when it is ready to proceed to the next stage.

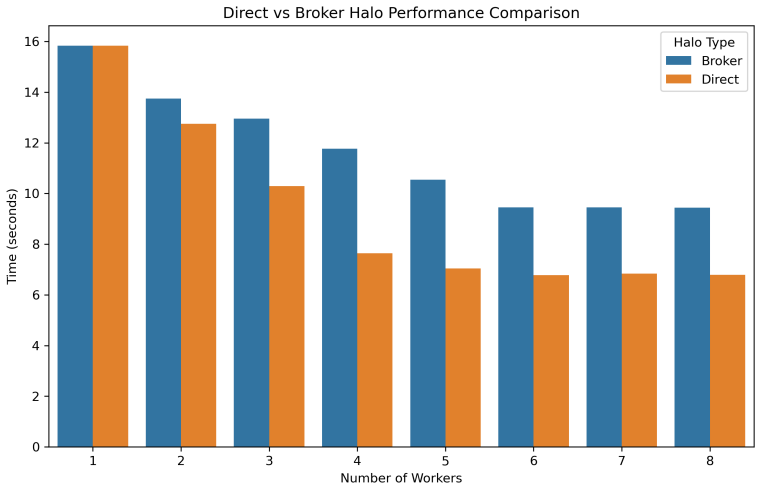

Figure 4: Performance between Broker-Server and Server-Server Halo Exchange

To improve this, we could implement a **Vector Clock** to keep track of the causal relationship between Halo Exchange processes on different servers to make it fully ordered. Moreover, we can use Singhal–Kshemkalyani's differential technique to further reduce the space complexity.

3.6 Resilience to Component Failure

Focusing the Key Press event on distributed systems, the general idea is when the controller captures a signal, it will make an RPC call to the Broker to request the behaviour of either sending a signal to `KeyPressChannel` and for `RunGol(..)` to pause, quit, close all components or loading from the `GameState` to save the snapshot of the world.

When `k` is pressed, the client makes an RPC to send a kill signal to notify the RPC goroutine `RunGol` to close all servers. Then, we set `GameResponse.Kill` flag to true for the client to make another call to close Broker after those 3 main RPCs have returned. It guarantees each component is closed while no other component is trying to make a call on a close connection.

For **Fault Tolerance** extension, we have set an attribute `Resume` flag on global `GameState`. When the client presses `q`, it will make an RPC to the Broker and the Broker will send a signal to `KeyPressChannel`. When `RunGol` receives a quit signal, it will acquire the **Mutex Lock** to prevent race conditions by other goroutines, then save the **GameState**, set `Resume` flag to `true` and tell all the RPC calls to stop. When a new controller is connected, it will similarly make an RPC to Broker to `RunGol`, but this time it will load the previous `GameState`, since the `Resume` flag is `true`, and toggle it back. To further improve, we can establish a **Broker Pool**. Even if one broker fails, the system can switch to another available broker to ensure the execution of Gol.