Concurrent Implementations of Game of Life

Will Bowden, Blaise Sheehan
yk22034, mq22858@bristol.ac.uk
University of Bristol

February 19, 2025

1 Parallel Implementation

1.1 Serial Implementation

We had an original implementation utilising no concurrent or parallel principles. This modelled a simple Game of Life simulation by iterating a 2D matrix of cells and changing the PGM value of each cell depending on the number of neighbouring alive cells. This was modelled using a nested for loop to examine cells in a 3x3 neighbourhood with a modulo operation ensuring proper wrapping around the edges.

1.2 Functionality and Design

In our parallel approach, we use a distributor to manage the distribution of tasks among multiple workers. To facilitate communication between workers, we define a channel for our updated world, denoted as `newFrames`.

For each computational turn, a goroutine is initiated using a `select` statement within the `calculateNewState` function. This statement is designed to identify when data has been received on the `newFrames` channel, signifying that the world's state needs to be updated and the turn has been processed.

Within `calculateNewState`, we dynamically set up a channel for each worker and calculate the bounds of the world that each worker will operate on. Subsequently, a goroutine is launched for each channel, starting the corresponding worker. The structure of our `worker()` function closely mirrors our serial implementation.

In the process of gathering each worker's updated slice of the world within `calculateNewState`, we assemble the updated frames of the world and send it down the `newFrames` channel into our distributor.

The distributor plays a crucial role in managing keypress events within the same `select` statement. This ensures that our world is never partially updated or updated after a key is pressed. Keypresses are predominantly handled through the `events` channel. Notably, the pause functionality employs a 'double' method. Upon calling `p` once, the system enters the helper function `handlePause`, which then awaits a resuming 'p' press within a `select` statement, effectively blocking further execution until the resuming pause command is received.

Note, we optimised our code using a `makeImmutableWorld` function, hence we only send a function as opposed to a copy of the world itself. This allows for a single copy of the world and likely reduces execution time with the worker goroutines.

1.3 Problems Solved

1.3.1 Dividing Board

- **Calculate Y Bounds:** The y bounds (`y1` and `y2`) for each thread are calculated based on the slice size. If `i` is the current thread index, the lower bound (`y1`) is given by $(i-1) \times$ sliceSize, and the upper bound (`y2`) is given by $i \times \text{sliceSize} - 1$. This ensures that each thread works on a distinct slice along the y-axis.

- **Handle Remainder:** If the current thread is the last one (`i == p.Threads`), a remainder is added to the upper bound (`y2`) to accommodate any uneven division of work among threads.

- **Reassemble Data:** After all workers have completed their tasks, the code receives the processed slices from each worker through channels and appends them to `newFrame` in the correct order.

This approach allows for a dynamic division of work among threads, accommodating both even and uneven distributions, enhancing the efficiency of parallel processing.

1.3.2 Ticker Implementation

We made a crucial adjustment to our ticker implementation during development to address data race problems and improve goroutine synchronisation. Our new code ensures the ticker and frame calculation are not intertwined in the same goroutine. This separation helps isolate variable usage and avoids potential data race conditions with multiple go routines. These changes collectively contributed to a more robust and synchronised implementation, removing data race problems and ensuring accurate handling of goroutines in our code.

1.4 Critical Analysis

All benchmarking within this report was done using inbuilt benchmarking, 5 counts and a MATPLOTLIB Python program. We also utilised Apple's CPU Usage Monitor and developed a CPU usage program for Linux.

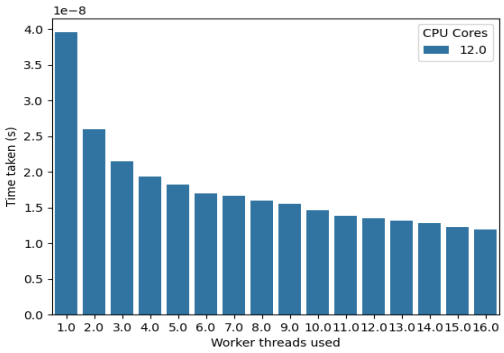

Figure 1: Intel 12-Core Parallel Test

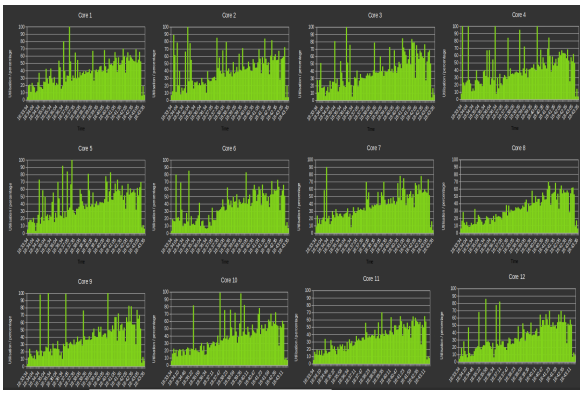

Figure 2: Intel 12-Core CPU Usage During Parallel Test

We observe a mean speed of 16.7 seconds and with increasing numbers of threads, we achieve a 4-50% decrease in time with more workers. A plateau occurs past the point of 12 (4-3% changes), likely due to filling the core - observe in Figure 2 our computation power is evenly distributed across each core. Increased parallelism lead to improved execution times, with additional workers plateauing at around 1.4s.

1.4.1 Extra Reading

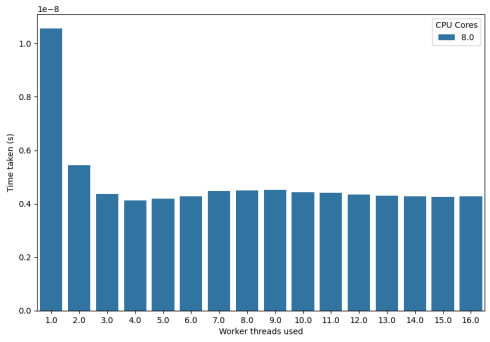

Figure 3: M2 Macbook 8-Core Parallel Test

We observed anomalous outcomes in this test, particularly the absence of an exponential/plateau trend across all 16 threads and a 10-fold decrease in time. This test was repeated five times, yielding similar results on each iteration. The highest speedup was with four workers, resulting in a

Speedup Value (S) of 2.45. This corresponds to an efficiency rating (E) of $E = \frac{S}{P} = \frac{2.45}{4} = 0.612$. Therefore, the maximum speedup per core is 0.612.

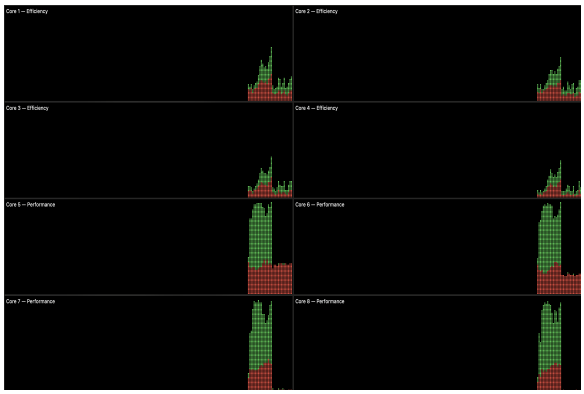

Figure 4: M2 Macbook 8-Core CPU Usage During Parallel Test

Our results can be explained by noting the M2 chip has 4 efficiency cores and 4 performance cores. These performance cores dominate computation until higher thread counts after which efficiency cores take some computation. The plateau seen in Figure 3 occurs when both core types contribute evenly. Note, the increase after 4 threads is caused by the visual decrease on the performance cores as in Figure 4.

2 Distributed Implementation

2.1 Functionality and Design

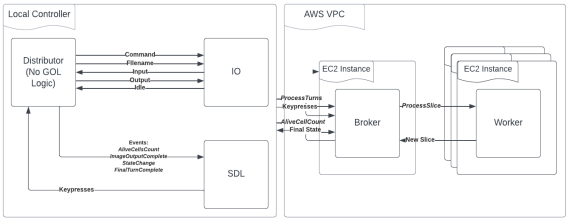

Figure 5: Distributed implementation diagram

We began our distributed implementation by turning the workers from parallel into RPC servers, and communicating with them from the distributor directly. This quickly proved to be difficult to work with, with too much coupling between the GOL logic and the local client. So, we followed the guidance and created a broker to operate in between the workers and the distributor. This broker is an RPC server in itself and listens for connections from the distributor, as well as managing connections to the worker nodes. All three types of node make use of values defined in `stubs.go` to express and use their interfaces.

The game is run when the distributor sends a `ProcessTurns` RPC call to the broker. The broker then calculates the bounds for each worker and sends a `ProcessSlice` RPC call to each of its n connected workers. The workers calculate and return their slice, where the broker reassembles the slices in order and then begins another round of calculation. The broker repeats this process until the desired number of turns have been executed, at which point the final state is returned to the distributor.

The broker also responds to RPC calls from the distributor that request `AliveCellsCount`, or request operations that correspond to keypresses the user has entered.

Note: Our distributed implementation can, theoretically, function with $n \leq$ `p.ImageHeight` workers. However, we only ran benchmarking tests on $n \in \{1, 2, 4\}$ workers. Running with more than 8 workers would likely begin to incur communication overheads that would make it unfeasible for real world use.

2.1.1 Data Transmitted

The data transmitted over the network in our distributed implementation is the following:

- `distributor` → `broker` sends: Initial world state and `Params`. Sent once at the start of execution.
- `broker` → `worker` sends: Current world state, `Params` and integer slice bounds. Sent at the start of execution of every turn.
- `worker` → `broker` sends: Slice of world state. Sent at the end of execution of every turn.
- `broker` → `distributor` sends: Final world state. Sent at the end of execution of

`Params.Turns` turns, when the client quits by pressing `q`, or when the client takes a screenshot by pressing `s`.

The broker also sends back integers for the cell count and turn number every time the ticker requests a cell count. The world state needs to be sent from distributor to broker to set up the initial state, and needs to be sent from broker to worker in order for the worker to calculate the new state of its allocated slice. Note that this is not the most efficient way to transmit the necessary data, and an improvement is outlined in **Potential Improvements** below.

2.1.2 Fault Tolerance

We've implemented error handling to address issues related to unreliable connections between the broker and various instances as well as listeners. Our system is designed to resume from the previous state after pressing the quit key. This functionality is achieved through the use of a quit boolean flag. During each call to process turns, if the quit flag is set to True, it means we're resuming from a previous client-quit session. This reduces errors due to use of global state and differing it from k-keypress. When controller terminates, we can resume handling using other components.

2.2 Problems Solved

2.2.1 Preventing data races in the broker

With many available RPC functions that could be called on the broker, we needed a way to ensure that the broker's state was not being accessed simultaneously. So, we added a Mutex lock to the broker's receiver interface to ensure that `g.state` was not being accessed simultaneously. This worked, but more problems arose when our implementation started occasionally failing `TestAlive` tests. Upon running the tests, the returned values would be wrong. We soon realised that this was due to the `g.turn` turn counter being accessed simultaneously. By wrapping all state access within a block of `mutex.Lock()`/`Unlock()`, we no longer failed `TestAlive`.

2.2.2 Allowing keypresses during a pause

Our previous implementation of pause functionality involved `Lock`ing the Mutex lock on the broker when it received the first `Pause` RPC call, and `Unlock`ing it on the second call. This worked to pause the execution, but prevented any other operations from being carried out on the broker, and didn't seem like an appropriate way to implement the pause.

Instead, we opted to use a state variable `g.pause` to track whether execution should pause, and busy-wait inside the `ProcessTurns` goroutine (but without locking the Mutex) while `g.pause = true`. This meant that other RPC calls could be handled and returned while the game state was not being evolved. I.e `AliveCellsCount`, and all keypress functions

2.2.3 Gracefully shutting down the system

Upon receiving a `k` keypress, we wanted our distributed system to shut down gracefully. Initially, we tried sending a `KillBroker` RPC call to the broker, which would send `KillWorker` calls to the workers. Upon receiving this call, the workers would send a signal down a `signal` channel, which allowed the `main` function to finish and the program to exit.

This however caused errors: `Error: send down closed network connection`. This was due to the workers trying to send results back to the broker, after the broker had already closed. To fix this, we added a waitgroup to both workers and the broker, which waited for all goroutines to finish before the server could shut down.

However, we were still experiencing the error. We later found this to be because the workers would finish their `KillWorker` routine, and then the RPC server would try and return the response to the broker. However, the RPC server on the worker would have already closed after the kill signal was sent down the `signal` channel. To remedy this, we broke up the `rpc.Accept()` function into its components so we could use the waitgroup to await connection hangups. We then ran `client.Close()`, first from broker → worker, then from distributor → broker, to hang up the network connections before the nodes were to shut down. This way, we no longer experienced any errors relating to shutdown.

2.3 Critical Analysis

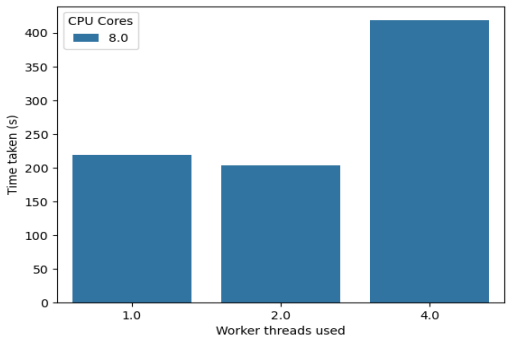

Figure 6: Distributed, with local broker

To conduct our benchmarks, we set up multiple t2.medium EC2 instances on AWS. The broker, running locally on our machine, would communicate with the worker nodes on these AWS instances via RPC to their public IP addresses. We ran 5 tests per benchmark.

We can see that all executions of our distributed system were significantly slower than our parallel version. This can likely be explained by the communication delay of the RPC calls between broker and workers. We can see a sharp increase as the number of worker nodes increases to 4. Again, this will be because our local broker is having to send 4 RPC calls to AWS (which was configured to run in the US-East region (North Virginia)), meaning the response delay was very high.

2.3.1 Distributed implementation with AWS workers and AWS broker

Opting to run benchmarks with an AWS broker within the same VPC as the workers, we yielded the following: This produced performance increases from an average of 260 to 36 seconds. This is likely due to network latency, hosting the components with the highest communication-overhead on AWS as opposed to locally avoids slower response times caused by multiple RPC calls and inherent latency in internet communication. We note that increasing the number of workers in this approach, improves speedup values (decreased thus improved execution time) when overhead is accounted.

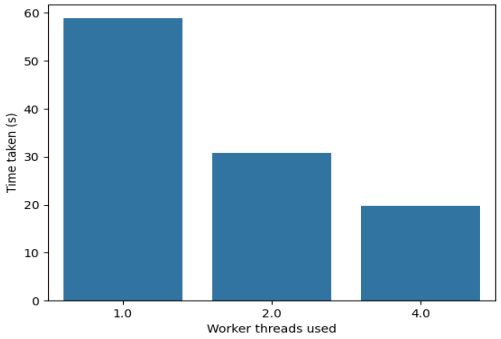

Figure 7: Distributed, with broker running on AWS

2.4 Potential Improvements

A substantial improvement may be achieved by implementing halo exchange. Our responses/requests between components include the entire state of the world at any one turn. This induces a lot of overhead, and considering that each worker only needs to know the cells surrounding its slice, there is redundant data communication. Halo exchange may optimise this by allowing the workers to communicate only necessary cells whilst only handling their slice.

3 Extension - Distributed SDL Live View

3.1 Implementation

We decided to implement the Distributed Live View extension. To do this, our execution follows the path below:

1. The distributor sends a `ProcessTurn` RPC call to the broker
2. The broker sends `ProcessSlice` RPC calls to the workers, along with the whole world state.
3. The workers calculate the cells that will have flipped, and returns nothing but a list of `util.Cell` objects
4. The broker concatenates the return values, and sends the complete list of flipped cells to the distributor, along with the current completed

turn count. The broker updates its internal world state with the workers' results.

5. The distributor then sends the `CellFlipped` events down the `events` channel to update the live view.
6. Steps 1 through 5 repeat `Params.Turns` times.
7. On the execution of the final turn, when the user quits, or when the user screenshots, the broker sends the whole world state back to the distributor for it to be exported as an image.

Our extension implementation still allows all keypresses to work as expected, and for an adjustable number of nodes (with AWS instances being setup manually).

3.2 Performance Comparison

In order to quantify and explain the performance overhead of the live view, we ran a benchmark test on it, with both the broker and the worker nodes on AWS. These tests yielded the following results:

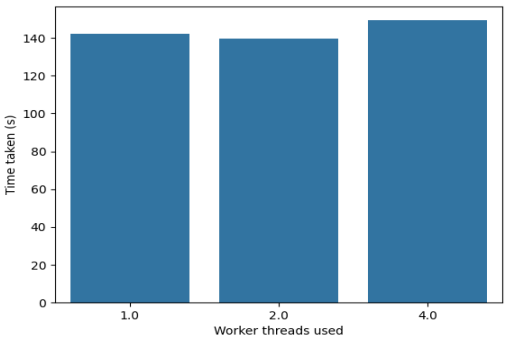

Figure 8: Live View Extension, all nodes on AWS

If we compare with Figure 7, we can see that the average execution (140s) is approximately 379% slower than the average execution time of the original distributed implementation (37s). This is likely due to the sheer number of RPC calls being made, one from the distributor and n (where n is the number of workers) from the broker **every turn**. The extension version made `Params.Turns` calls to a US AWS region, whereas the distributed implementation (with remote broker) made only 1. This massive communication frequency has drowned out any performance gains we might notice, and the performance of the system in testing was massively dependent on internet connection speeds.

To further emphasise the overhead added by the frequent communication, we tested our extension with all nodes running on the same machine, so as to minimise communication delay. This resulted in the following graph:

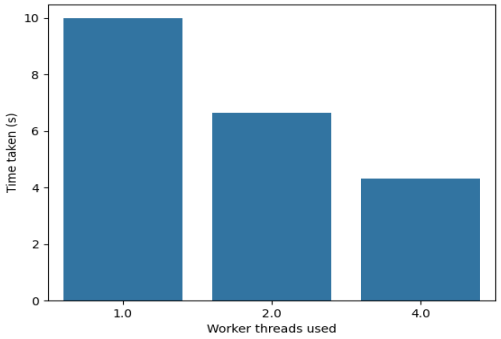

Figure 9: Live View Extension, all nodes on local machine

Here we can see that the extension was lightning fast, executing 1000 turns in an average time of approx. 7s. This shows how severe the delay that was being imposed by the frequent communication was. In both this test and the distributed test that gave Figure 7, all nodes were running on the same network, but the extension version was still significantly faster than the distributed. This can likely be explained by a few things:

- The AWS nodes communicated within the same VPC & subnet, but were on different machines, so had a slight but nonetheless larger communication delay than those on the same machine.
- The worker nodes in the extension only communicated cells that *needed changing* back to the broker, rather than sending their entire slice as the distributed version did. This means much less redundant data was being sent back and forth repeatedly, which likely decreased processing time.