A parallelised and distributed approach to implementing Conway's Game of Life

Aaron Chan
University of Bristol
ho21739@bristol.ac.uk

Ferdinand Hubbard
University of Bristol
ej21378@bristol.ac.uk

December 6, 2022

Abstract

We intend to explain our two solutions for implementing Conway's Game of Life in Golang, namely the Parallel and Distributed versions. Within this report, we will discuss the impacts of thread usage in both our implementations, as well as the effect of distributed computation as opposed to single-machine computation. In essence, we found that execution time was sped up as we increased thread count, reduced branch instructions and implemented a halo exchange mechanism.

I. Introduction to the parallel solution

As an initial attempt, we began by implementing a single-threaded Game of Life (GoL) Engine, expanding upon this work with parallelisation by virtue of concurrent go-routine workers. Recognising that there exists various methods to calculating the perimeter for `getNeighbourCount`, we tried various counting methods - taking into account the modularity of Game of Life - as well as finding channel alternatives.

II. benchmarking methodology

Our means of evaluating parallelisation performance gains was via the Go testing framework. Using 512px x 512px PGM images, we repeated each benchmark 10 times, taking the average runtime for each thread to acquire the central value, and plotting them. Each feature implementation was benchmarked in the same fashion, using Linux lab machines with CPUs being Intel i7-8700 (6 core, 12 threads).

III. A split personality program

A. Parallelisation logic

The baseline implementation's distributor function starts with reading the image into 2D slice via *io* commands, before evenly dividing the workload between workers. With the aid of go-routines, workers have

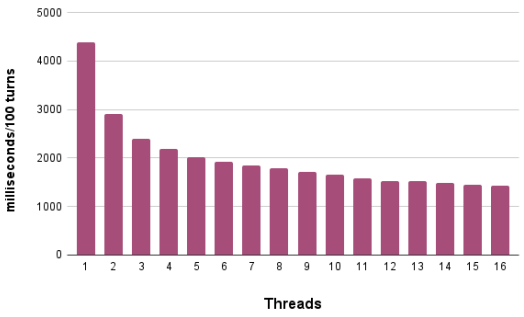

Figure 1: *Modulo Implementation*

access to the old slice as well as the boundaries in which to process. By using `getNeighbourCount`, we apply the Game of Life laws onto the world, piping results into a blank slice of size:

$$(endRow - startRow) \times ImageWidth$$

Sending the results back to the distributor, we integrate back together the new slices, move the pointer to the old world to this newly-merged world and send the appropriate signals on the *events*, *io* channels.

B. On the topic of Analysis 1

From Fig. 1, we can conclude that multithreading have an overall improvement on performance. There is a correlation between increasing number of threads and decreasing execution times, likely due to some functions, such as `getNeighbourCount`,

being "embarrassingly parallelisable". We favour the term "delightfully parallel". As we have more cores for processing, the time taken for each turn is reduced, hence the curve.

We do take notice of the obvious trend of diminishing returns - the gradient of the time-thread curve flattens. We attribute this to unparallelisable core tasks such as IO - the major slowdown that is reading and writing to/from the disk, and submit that using more threads than a CPU physically has causes overhead in scheduling, minimising the benefits of allocating extra threads for processing. We suspect also that, contributing to this, the workload division does not change as much the more threads there are, which explains the reciprocal graph shape curve.

Regarding the numerical results, we find greatest performance was with 16 threads (despite the diminishing returns) with a 67.6% decrease in processing time. As for the worst change in performance thread-to-thread, we see the 14th - 15th threads having the smallest difference in performance change (0.353%), but acknowledge the potential anomalous data point.

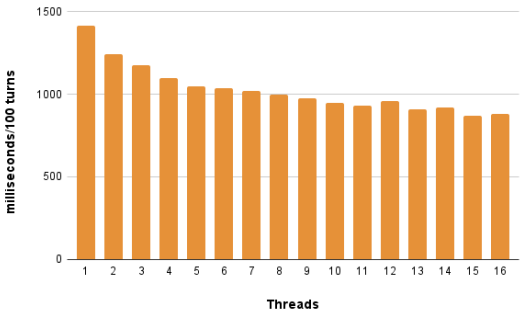

Figure 2: *Branching Implementation*

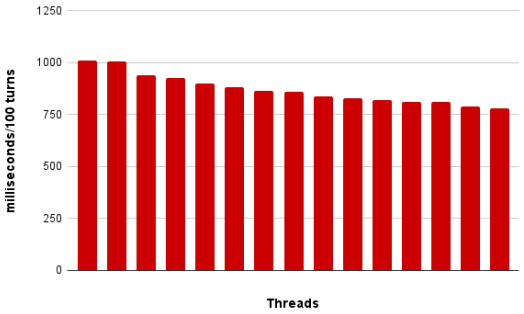

Figure 3: *Bitmask Implementation*

IV. How to count?

A. Counting methods

It is well documented that division and other non-atomic arithmetic operations are slow. Thus, we submit our alternative implementations for `getNeighbourCount`. Fig. 1, our original implementation, uses the modulo operation (%), whereas Fig. 2, Fig. 3, use Branching (if-statements) and Bitmasking respectively. From this data, we see that Bitmasking is the fastest on average, but we can identify positive notes from all three.

B. On the topic of Analysis 2

Statistically-speaking, we find the solution using modulo operations is most affected by thread usage, having standard deviation 413ms, as opposed to 107ms and 73ms for branching and bitmasking respectively. Interested persons will find our results achieved by:

$$SD(\forall x \in runtimes : max(runtimes) - x)$$

As discussed before, this, however, implies not that higher threads means greater performance as there exists other, non-parallelisable tasks setting a base level of execution time. The low standard deviation of the bitmasked implementation alludes to this portion of the program being as near optimised as we possibly can.

When it comes to average times for threads, bitmasking takes the lead with 890ms, followed by 1026ms and 1989ms for branching and modular respectively. This does identify branching for the greatest decrease in runtime, with a 74.2% decrease, as opposed to 32.1%.

C. Our Algorithms

Code-wise, our initial implementation for `getNeighbourCount` used modulo operators unnecessarily. It did the following:

1. Set row = coordinate row % ImageHeight
2. Set col = coordinate col % ImageWidth
3. If row | col < 0, then add ImageHeight/Width respectively
4. If world[row][col] was 255, increment alive count

With this, while it succeeds in its purpose, it does not succeed efficiently. Areas of improvement include removal of the modulo operator, which, here,

does not serve any real purpose. This leads to our second implementation - branching:

Algorithm 1 Branching getNeighbourCount

```
Let i, j ← (row, column) on world
Let O ← [{−1, 0, 1} × {−1, 0, 1} − {0, 0}]
Let alive ← 0
for (x, y) ∈ O do
    row ← (i + y)
    if row < 0 then
        row ← ImageHeight − 1
    else if row = ImageHeight then
        row ← 0
    end if
    col ← (j + x)
    if col < 0 then
        col ← ImageWidth − 1
    else if col = ImageWidth then
        col ← 0
    end if
    if world[row][col] = 255 then
        alive + +
    end if
end for
return alive
```

Although the line count has certainly increased, the performance has drastically improved. Citing the 74.2% decrease, this does appear to be "good enough" as an implementation. An improvement may be to rid ourselves of branches, as this involves branch prediction which itself is intensive, due to how the CPU does not know which path will be taken ahead of time. If we use only logical operations, we get the Bitmasking implementation:

Algorithm 2 Bitmasked getNeighbourCount

```
Let i, j ← (row, column) on world
Let O ← [{−1, 0, 1} × {−1, 0, 1} − {0, 0}]
Let alive ← 0
for (x, y) ∈ O do
    row ← (i + y)&(ImageHeight − 1)
    col ← (j + x)&(ImageWidth − 1)
    alive ← alive + world[row][col] >> 7
end for
return alive
```

This algorithm is our fastest implementation. Minimising both line count and runtime, it counts the neighbours using purely bitwise operators and assignments - no branching needed.

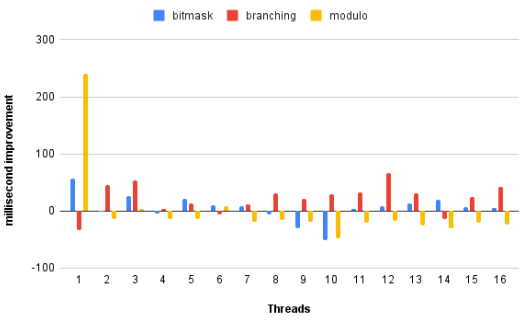

Figure 4: *Memory Sharing*

V. Memory Sharing

A. Motivation

Looking at our achievements in counting, we noticed overhead associated with channel communication. Exploring Go's implementation for channels, we see much overhead. Since we know a channel is a thread-safe memory sharing mechanism, we can rid the overhead, using only mutex locks and condition variables.

B. Implementation

Our homemade channel implementation does the following:

1. Acquire the mutex lock
2. While `condition` is false, await broadcast unless non-blocking send
3. Use the value, then broadcast to other processes
4. Finally, unlock mutex

We introduce 2 methods, `Send` & `Receive`. `Send` waits for the memory space to be empty, and replaces it with a value. `Receive` waits for the memory space to point to a value, and then replaces it with `nil`. `Go 1.12` does not have generics, which vastly complicated implementation. Benchmarks with memory sharing (and bitmasked `getNeighbourCount`) is shown in Fig. 4.

C. On the topic of Analysis 3

This data was not very analysable. The lack of clear trend and sporadic data points suggests no real difference in performance when adding memory sharing, or faulty implementation (which we realistically do consider a possibility). A trend, if any, we identify

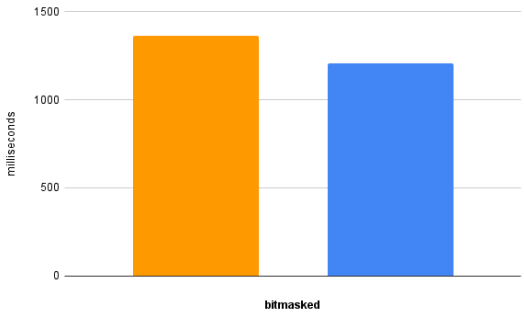

Figure 5: *1 worker vs. 1 thread*

is that the Modulo performance suffered, but due to the lack of clarity, this is inconclusive. We add that repeated trials yielded similar style results. Our method used the following:

$$\forall t \in T : ORIG[t] - MEM[t]$$

Where MEM is memory sharing implementation time, $ORIG$ is channel impl. time, T is the number of threads.

VI. Distributed Implementation

Similar to how we did the parallel side, we started the distributed section by using a single worker, this time with the worker being a separate process and communicated to via Remote Procedure Calls (RPC henceforth). This expanded to doing using multiple workers, implementing fault tolerance and finally halo exchange.

VII. A lonesome worker

A. Logic

Our single-worker distributed implementation moved much of the processing to said worker. From the distributor, we sent a single RPC call to the worker, processed GoL turns on the worker, then sent the final state back. Results are shown in Fig. 5.

B. On the topic of Analysis 4

We decided to test single threaded parallel version against a single worker distributed GoL. Seeing that the 1 worker ran with runtime 1362ms versus the single-threaded 1208ms, we concluded that the 154ms difference was due to the latency between Bristol and AWS London, in addition to some potentially contributing overhead from RPC calls. The rest of the runtime we concluded was the natural runtime of the worker processing Game of Life.

VIII. Friends? - Multi-worker GoL

A. Implementing multiple workers

With the introduction of multiple workers, we needed to introduce a broker to interface the workers for the controller and vice versa. This did require moving turn iteration logic to the broker, requiring a major refactor. In the end, the system revolved around the broker with bidirectional RPC communication betweeen workers and the controller. We equally divide work between workers, and all processes keep an internal state of some sort. By massively simplifying, the broker does the following:

1. Listen for worker and controller connections
2. On connect, call the worker/controller back (bidirectional)
3. Listen for GoL process request from controller
4. Iterate, dividing and sending work to each worker.
5. After getting all parts back together, apply changes to broker internal state and divide new state up

In order to call the worker back, this meant that the worker needed to provide its own IP address - the same process happens for the controller. The reason why the worker needs bidirectional communication is that later on with halo-exchange, we will need this to push the state when the turn iterator is at the worker, and also to allow workers to disconnect without errors.

When it comes to the reassembly, the worker sends the flipped cells to the broker, and because no worker interferes with another worker's section, we can just apply the changes in any order to the broker's internal state.

B. Analysis 5

Shown in Fig. 6 is the distributed system when run on a local machine (due to port-forwarding issues). We can see a general negative correlation between workers and runtime - as workers increases, the runtime decreases. Noticing that the runtime begins to incline at the end, we initially thought this was due to

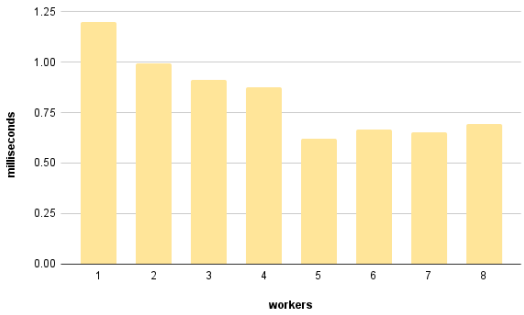

Figure 6: *Multi-worker local performance*

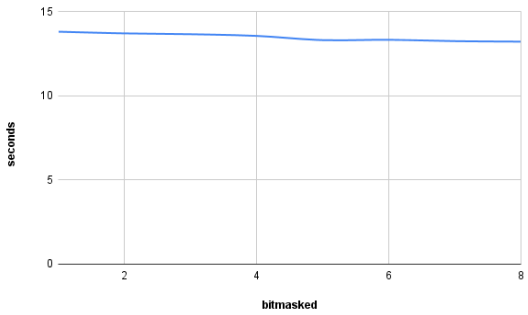

Figure 7: *Multi-worker AWS performance*

RPC overhead, but it may have been due to how running on 1 machine starts to cost more performance than it improves.

IX. Tolerating faults

In our previous implementation, we never checked for worker disconnects. This leaves us in a vulnerable position - if a worker were to be terminated at any point, then we would have one section of changes missing. This would show itself with a static portion on the SDL window. There is also the case of adding more workers. Either way, it means we need to keep the order of workers. We present Algorithm 3 to solve this.

Algorithm 3 Worker-diff checker

```
Require ids ⊆ ℕ
Require oldIds ⊆ ℕ
Let idMap ← Map
for id ∈ ids do
    idMap[id] ← addr
end for
for id ∈ oldIds do
    if idMap[id] = nil then
        return False
    end if
end for
return True
```

What this algorithm achieves is detection of disconnects and reconnects. Given that we have the loop invariant (in turn iterator) "no two workers will have the same worker id", we can use this algorithm. If we detect a change, recalibrate the workers to adjust for new/missing worker. One may ask, does a worker disconnect not throw errors when calling RPC methods? In this case, we utilise `client.Go`, which notifies the calling process of errors without crashing. So, if error, then trigger recalibration by resetting each worker's internal state.

More specifically, the broker sends an RPC call to each remaining worker with a fresh state from which to evolve from. The previous internal state of each worker is discarded, even if it is one turn ahead as we use the broker's internal state, which is only pushed to if all workers return their flipped cells fine.

X. Implementing Halo Exchange

A. How Jesus Christ came to be

Our first attempt wasn't exactly the definition of halo exchange. It used halos distributed by broker to each worker, who then used the halos to update their internal state. Now, our implementation transfers the turn iterator logic to the workers. The sequence of events happens as follows:

1. Worker connects to broker, and is initialised with parameters of controller.
2. On initialisation, worker will dial the top and bottom neighbours (if any), then await work.
3. On work, send its halos, and evolve its own slice given the neighbouring halos
4. Results are pushed to broker and loop

This oversimplified view of halo exchange is represented by a significant amount of implementation code. We present our results with Fig. 6.

B. Analysis 6 - Judgement Day

The absurd difference in value between Fig. 6, Fig. 7, was due to what we identified as transmission latency. The broker was positioned in `us-east-1`, North Virginia, and the 7000km+ differential caused

the latency to be very high. We did notice, despite the glaringly obvious latency issue, that the time did start to decline as more workers were added, implying that there may have been some speed increases due to our distributed system. However, we do not reject the notion of these being oscillations in latency values.

Giving the benefit of the doubt, had the EC2 instances been set up in the correct region, then we should have noticed a significantly smaller runtime with a lower standard deviation - that is, less variance in data points.

XI. Areas for improvement

- Memory sharing implementation for channels - we were able to create an alternative to channels, but this ended up more or less the same performance as a regular channel, if not worse. A strong area of improvement would be to rewrite this, and substitute all the other channels used with our implementation
- Parallelising workers - we did not manage to implement this in time. Since all the code for multithreading was in our parallel code, we could have imported the code then implemented it inside our workers.
- Controller on AWS node - we found issues with latency when using AWS nodes, so if the "local" controller was localised in the AWS data center, then we would eliminate this issue. We, however, ran into issues with SDL not working on EC2 machines and port-forward which meant these latency issues still happened.
- Broker fault tolerance - As a extreme extension, we have the broker as our single point of failure. If the broker fails, then nothing can connect and no processes can occur. We could, in theory, create a second broker to be redundant in case the instance running the initial broker fails.

A. Thanks

- Lecturers and TAs for their support
- Monster Energy™
- My missing sanity