An Implementation of Conway's Game of Life in a Parallel and a Distributed Scenario with saving and quitting while paused

Gabriel Galadyk
us21186
Department of Computer Science
University of Bristol

Nikhil Parimi
ob21372
Department of Computer Science
University of Bristol

I. Introduction

Presented below are two different implementations of Conway's Game of Life in the Golang Programming Language; the first is a parallel implementation, where all work is performed on a single machine using multiple worker goroutines, while the second is a distributed implementation, where all work is performed using multiple AWS instances where the Game of Life logic is stored.

We will examine the implementation methods we have used in detail, suggest possible improvements and strategies that would have improved the run time of the program, and eliminate bottlenecks in the program that slow it down. Finally, we will decide whether for the purposes of running Conway's Game of Life as efficiently as possible, whether the parallel implementation or the distributed implementation is more suitable.

II. Parallel Implementation

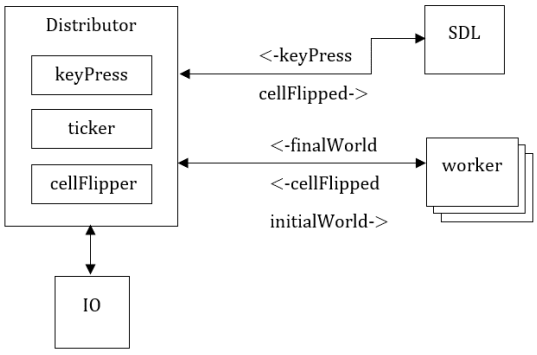

Fig. 1: Diagram of the final parallel implementation

A. *Development and our Goal*

The first step in development was a rudimentary and sequential Game of Life implementation that operated on a single worker. This single worker implementation, having successfully passed initial trials by computing the Game of Life, allowed us to with confidence engage with parallelising the work among multiple workers. As part of the task, the parallel implementation was updated to operate with a maximum of 16 different worker threads, in order to accommodate for this, the inclusion of mutex locks and helper functions to assist with accurate distribution of the world were implemented, alongside that, functionality for responding to key presses was created, with the ability to save and quit while paused introduced as well.

Our overall goal and intention of the implementation was to ensure the program is as concurrent as possible and multiple extra goroutines were introduced to achieve that goal. Our overall program therefore consisted of the following:

- *distributor*, called at start along with *io* and begins execution by first loading the image from *io* into a world. It then instantiates the channels for communicating with the workers and the *cellFlipper* goroutine, it calculates the bounds next, and instantiates the workers, before initialising the *cellFlipper* goroutine, the *keyPress* goroutine, and the *ticker* goroutine; all are anonymous within *distributor*. Finally, the game of life computation is run, where data races are avoided using a mutex lock on the turn, and current world.
- *cellFlipper, ticker,* and *keyPress;* anonymous goroutines that handle events that can happen during execution. In different implementations, *keyPress* and *ticker* were removed in favour of including them in the *for-select* statement while cellFlipper was retained. *cellFlipper* received cell data from the workers for cells that needed to be flipped in SDL, while *ticker* and *keyPress* handled their own events respectively.
- *worker;* which handled actual Game of Life logic, processed a single turn of a given world and communicated back and forth with distributor. It also had access to the channel to communicate with the *cellFlipper* as within the worker was where the data about cells that got flipped was contained.

Finally, upon termination, whether natural or by keypress, the distributor would save the final image of the world, then gracefully exit, which then down the line results in main terminating and all other goroutines exiting.

B. *Difficulties and Optimizations*

During development of the parallel implementation, we have progressed towards our goal in steps and identified multiple sections and issues with how the implementation ran, as not all solutions to these issues

were implemented in time, we have settled on benchmarking two of the implementations we developed.

The main optimizations we have made which are the subjects our benchmarks are:

- **Handling of events interrupting execution;** Initially, we have used a *for-select* statement to handle events like keypresses or the demand for alive cells at the beginning of the turn, this however meant that such events had to wait for the end of the turn, and could not run otherwise. To align with our goal of concurrency, we have switched to using goroutines and a mutex lock which we will benchmark
- **Division of labour between workers;** During development of a naïve multi-worker implementation, we encountered an issue with dividing work with non-base 2 number of threads, our solution to start was to simply offload the remainder of work onto the last thread. Later on, we introduced an algorithm to properly divide the work as evenly as possible between the workers.

There were multiple other issues and areas for improvement we have identified as possible optimizations, these are going to be listed in section 3.D

III. Benchmarking Parallel

A. *Methods Applied*

We have performed our benchmarking on the Linux lab machines of the Merchant Ventures Building, the machines have an Intel i7-8700 with 12 logical cores running at 3.20 GHz. All 4 of our benchmarks were ran 20 times for 1000 turns on the 512x512 image with the average taken as the final result. The event handling was mainly stress tested by the ticker goroutine which exists as part of the specification for the program.

We have also performed one additional benchmark on the 5120x5120 image with our final implementation where instead, to observe how long it takes to perform a single turn, we have counted the amount of *AliveCells* events that were sent. As those are sent basically every 2 seconds on average, we deemed it suitable to count the rough duration of a turn.

B. *Results*

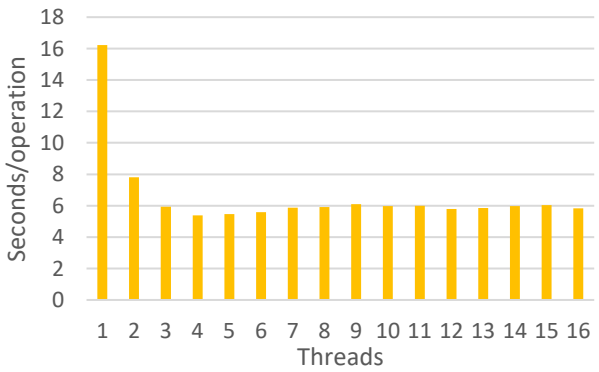

Fig 2: Result of the first parallel implementation; max concurrency, even distribution, 1000 turns

Our final implementation that we have chosen as our completed product, has resulted in the expected slope of drastically lowering the seconds/operation before averaging out as more threads are added. As seen in Fig. 2 our fastest runtime occurred with 4 threads, where the time was 5.3819 seconds to compute 1000 turns of GOL. The fact that the lowest time was reached in 4 threads could be the result of our usage of channels to communicate all information, alongside the fact that the entire world is being passed along said channels, it massively adds a lot of overhead and thus the speed up in computation is countered by the slowdown in communication. If a more efficient method for communication was introduced, we believe that we would obtain a more desirable graph that reaches the lowest point with more threads, rather than with less.

We have compared the results of our final implementation with the 3 other variants, they are going to be referred to as the following:

- GRJ – our final implementation, max concurrency, even distribution
- FSOJ – *for-select* statement, even distribution
- FSJ – *for-select* statement, offloading work to last thread
- GROJ – max concurrency, offloading work to last thread

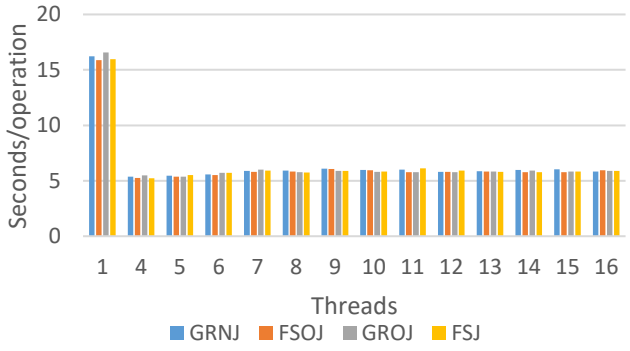

Fig. 3 Graph comparison of all 4 implementations, 1000 turns

The result of our benchmarking produced shocking results. It can be seen in the graph in Fig. 3 that the average performance of the implementations that used *for-select* statements produced faster running times than the implementations that did not, including our main implementation, GRNJ. Despite the benchmarks only relying on the ticker goroutine to be the primary cause for any runtime difference, it has indeed produced results that let us confidently say that were we to run these benchmarks again but introduce periodical saves and pauses, these results would favour the FS implementations more; our original design goal of maximising concurrency has proven to not be as feasible and viable as we have thought.

Another oddity that was identified was the fact that for the first 7 threads, FSOJ, using the inferior implementation of offloading the work on the final thread, was always faster than FSJ. We believe this anomaly is caused due to

the last thread consistently having less alive cells to work on than the other threads (more on that in section C), thus attempting to divide the work evenly is not worth it in this specific case.

Fastest and slowest times	Implementation			
	GRNJ	*FSOJ*	*GROJ*	*FSJ*
Fastest	16.2287	15.883	16.5582	15.9796
Slowest	5.3819	5.2475	5.47988[a]	5.22383

[a] GROJ's fastest time was on the 5th thread

Fig. 4: Fastest and slowest time comparison

Finally, one last observation as showcased in Fig. 4, is that all the implementations except for GROJ reach the lowest time at 4 threads, before increasing again, GROJ instead reaches the lowest time with 5 threads at 5.37173 seconds, this conclusion was repeated when another 20 tests were performed on GROJ, this anomaly can only be theorised to be caused by the same concept as in the previous one, that the calculation to divide the world evenly takes longer than just having the last thread process the cells.

C. *Results On Turn Duration*

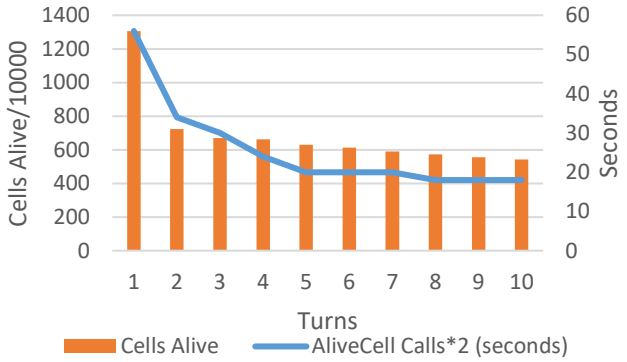

Fig. 5: Comparison between number of alive cells and duration of a turn

As mentioned earlier, there seems to be a clear relation in Fig. 5 between the number of alive cells and how long it takes for a turn to process. We believe this could be the result of how the workers handle writing to a new world, since in Golang, the default value for an int is 0, and a dead cell is also defined as 0, this results in *calculateNextState* being able to omit the write to the new world of a cell that is dead, and is meant to stay dead, this does not happen with alive cells; this could be the explanation for this slowdown, as mentioned before, it could also be the reason for FSOJ having an anomalous faster time than FSJ.

D. *Final Words on Parallel*

To summarise, our parallel implementations and the benchmarking provided exactly the opposite results we thought we would get. The creation of extra goroutines that handled the ticker and keyPresses (so that the GOL logic can run concurrently) did not produce the faster results that we thought would we would get. There appears to be merit in forgoing the process of making GOL as concurrent as possible and accepting some non-concurrent features like using a *for-select* statement instead.

There were also improvements that could have been made on parallel that could have improved overall running time, and even push the fastest time to occur when more threads are used, we identified these bottlenecks as:

- **Passing the entire world to the workers**; this most definitely caused a significant amount of lag during computation, and implementing something like Halo Exchange or a single global world instead of multiple copies would have improved execution
- **Slow Game of Life logic**; our current implementation includes a slow calculation for wrap around which could be improved to save on time and computation and further lower processing speed.

Overall, we are happy with how our parallel implementation turned out as well as the results it produced.

IV. Distributed Implementation

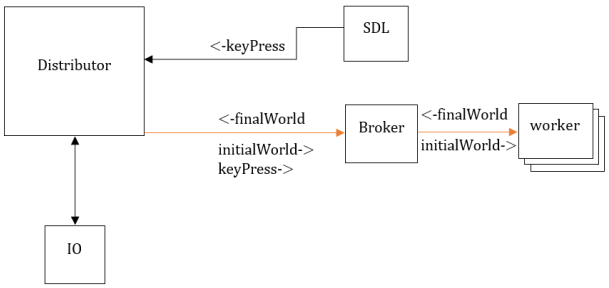

Fig. 6: Diagram of distributed implementation

A. *Development and our Goal*

Our methodology for our distributed implementation closely followed our parallel implementation. We sought to closely mimic the implementation and as such, apart from how the different components communicate with each other, we developed an implementation that achieved that goal.

Whether this goal was correct for overall speed of computation of the Game of Life in a distributed scenario was not considered, we have instead chosen to aim for comparing the parallel and distributed implementation together by making them similar, however we have identified and documented the issues that the distributed section has that could have accelerated the computation to a desirable result.

Note however, that the distributed implementation does not feature an SDL GUI view, the only purpose of the SDL window is to communicate key presses to the distributor, this has been reflected in Fig. 6 by a one-way arrow. Furthermore, one-way RPC calls are identified by the orange colour.

The overall implementation therefore consists of:

- *distributor;* also known as the local controller, it first loaded the image required from *io* and dials up the *broker* instance which exists locally on the same machine, it then calls the *broker* with the world and awaits a response, as it does not perform Game of Life logic, the *distributor* can handle keyPresses and the ticker concurrently without additional goroutines. When it receives a response from the broker, it proceeds with saving the image and terminating
- *broker*; acting as the main communicator between it and the workers, it loads the image into a global world, alongside the global turn, which allows other procedure calls into it to respond concurrently while logic is going on, same as with parallel. It initialises a hard coded amount of worker threads, and proceeds with Game of Life logic on the workers as normal.
- *worker*; this component exists on the AWS instances, and performs a single turn of Game of Life logic before responding to the broker to then be called again the next turn.

The different components all communicate using different response structs, called *stubs*, which all communicate different data, including communicating no data.

V. Benchmarking Distributed

A. *Methods Applied*

While not much data could be obtained from benchmarking the distributed implementation, it was nonetheless attempted.

Just as the parallel implementation, it was run on Merchant Ventures Building Linux lab machines, with the *workers* running on *t2.medium* instances on the AWS cloud services. The benchmark was ran 20 times per number of workers for 100 turns on a 512x512 image.

B. *Results*

Fig. 7: Benchmark of distributed implementation

The results of the distributed implementation showed roughly what was expected; the massive cost of transferring the world every single turn to the workers created a lot of overhead that slowed down the overall implementation even if the calculation of the world was indeed fast. There is a small drop in time taken on the second thread, but afterwards it continually rises, and we believe that beyond 8 worker instances it will continue to rise.

C. *Final Words on Distributed*

Unlike the parallel implementation, the distributed implementation would have benefitted more from optimization methods that were not tried before, for example, lowering the amount of data sent with a Halo Exchange or a shared distributed memory would have greatly benefitted the workers and would greatly alleviate the amount of data the RPC calls were sending at any time, furthermore, there could be room for making SDL graphics work with the distributed implementation, but we believe it would introduce more bottlenecks with even more RPC calls being made.