Concurrent Computing
Game of Life

Razvan David (oh18767)
Stefan Pruna (mc18112)
December 4, 2019

1 Introduction

Game of Life is a zero-player game, it requires no further input other than the initial state of the board. It follows a set of rules with the help of which the board is evolved to the final state. A cell's evolution is determined only by its neighbouring cells and its own state. This means that a Game of Life simulation can be heavily parallelized. This report aims to explore the performance differences between implementations of Game of Life in Go, with different levels of parallelism.

2 Functionality and Design

A *worker* is a single-threaded function that simulates Game of Life on a part of the board. The *distributor* is a function that controls the workers and sends them the board. A *halo* can be defined as one of the two extra rows of the board that a worker needs in order to compute the next turn, the row of cells before the first row on which the worker simulates, and the one after the last computed row.
In order to compare the performance gains of parallel computation applied to Game of Life, we implemented simulations using different levels of concurrency. Starting from no concurrency at all, all the way to a multi-server implementation using network sockets over which halos are sent.

2.1 Different levels of parallelism

2.1.1 Sequential and naive parallelism

While in the sequential implementation of the simulation, a singular worker evolves the whole board, in the naive parallel version, multiple workers evolve the board. In this naive implementation, the workers send and receive the board and the halos to and from the distributor at each turn. To make the program more efficient, worker functions are only called once, at the beginning of the simulation. Data is sent and received between the workers and the distributor via byte channels. The work is divided evenly between the workers, such that the maximum size difference between any worker's part of the board is 1 row. Our implementation supports any number of workers.

2.1.2 Improved parallelism

In the naive parallel implementation, the distributor receives and sends back the board from and to the workers at each turn. This is a bottleneck, as a worker has to wait for the distributor to collect the new board rows from other workers, reconstruct the board, and send the new rows back, all sequentially.
The optimisation comes from the observation that workers only need to receive the halos with the current board state in order to process the next board state. As such, the distributor is only needed to send the initial board to the workers and to process user interaction.

2.2 User interaction

Users can interact with a simulation by pausing it, saving the current state of it to file, or stopping it. The number of alive cells on the board is printed every 2 seconds. This means that for all implementations, some form of control and synchronisation of the workers is required.

2.2.1 Parallel interaction

In the sequential and naive parallel implementations of Game of Life, the workers interact with the distributor on every turn. For better performance, in the improved parallel version, apart from sending the initial board, the distributor does not interact with the workers at all. The workers can continue running on their own, and if they finish before being interrupted by the distributor, they will send the board back. On an IO event such as pause, the distributor synchronises the workers to the same turn. It does this by querying them for their current turn, then telling all of them to pause at the maximum turn any of them has reached. For the world to be saved, the distributor must first synchronise and pause all the workers, then request the world. On a timer event, the distributor synchronises the workers again and requests the number of alive cells from them.

3 Performance

After running a set of 10 benchmarks on each implementation and recording results, we observed a significant decrease in performance when comparing the naive parallel implementation to the sequential implementation. As it can be seen in Table 1, the naive parallel simulation performed about 2.85 times slower than the sequential simulation. In contrast, the improved parallel version performed about 1.49 times faster than the sequential program.

Table 1: Performance of different implementations

Implementation	Mean time	Standard deviation
Sequential	37.14s	1.54s
Naive parallel	105.73s	1.92s
Improved parallel	24.94s	1.22s

3.1 Explaining results

3.1.1 Sequential and naive parallel

The degraded performance of the first parallel implementation of Game of Life we benchmarked is caused by a bottleneck in the distributor. The distributor receives, reconstructs and sends the board back to each worker on every turn. The workers can start computing the next turn when they receive the data from the distributor, but this data is sent to the workers in sequential order. By the time the second worker has received the needed data, the first worker is almost done with computing its part of the new board state, as the computation itself does not take a significantly longer time than the transfer of data over byte channels. That being said, by the time the distributor has sent the data to all the workers, a large portion of them are already done and ready to send the data back. The sequential simulation is faster because the single worker that computes the new state of the board always has access to the whole board, and it doesn't need to send or wait to receive data over channels.

3.1.2 Improved parallelism

After the distributor has sent the workers their part of the initial board state, they only require the top and bottom halos in order to compute the next state of the board. The halo data that a worker needs in order to continue computation is computed by another worker. In the naive implementation, this data, but also the rest of a worker's section of the board, is sent to the distributor and back each turn. To optimise the naive implementation, the halos are exchanged from one worker to the other, over byte channels, such that no data is needed from the distributor, except for computing the first turn.

Quantifying parallelism To further analyse the difference between the naive parallel implementation and the improved parallel implementation, we ran 10 benchmarks on the 512x512 board. We measured how the performance of our solution scales with the number of threads (workers). The results are shown in Figure 1. The performance has been normalised such that for each implementation, the fastest result is 1. It can be seen that the naive parallel implementation does not scale at all, while the improved parallel implementation scales.

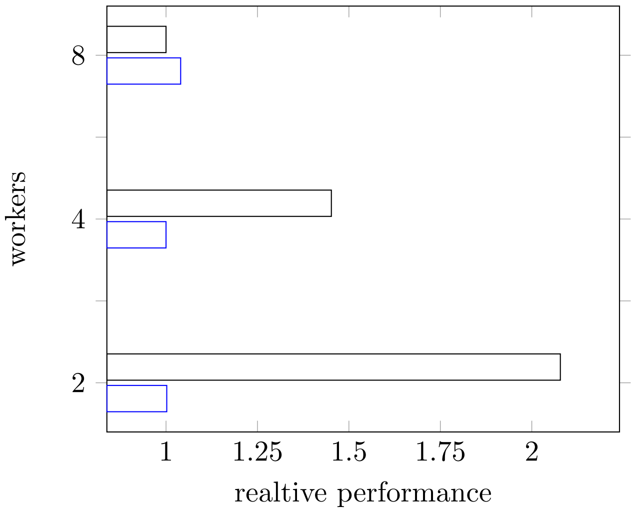

Figure 1: scalability of parallel implementations

User interaction In the improved parallel solution that uses the halo exchange idea described previously, user interaction can also be implemented in two ways, one slower than the other. The slower idea was for the distributor to send a signal to each worker on every turn. If there are no interactions, the distributor would send the workers a *continue* signal. For pausing, the distributor would send them a *pause* signal, and so on. This is slow, as the workers have to tell the distributor they have completed a turn and then wait for the distributor's signal. We improved this slow version by only sending workers signals when there is an event other than *continue*. By not synchronising workers at each turn, they can be computing different turns at some point in time. This problem was solved by synchronising them to the maximum reached turn, as explained in section 2.2.1.

3.2 General optimisations

By using CPU profiling, we were able to conclude that the initial implementations of two of our functions were performing very badly. The first function computed the modulus of negative numbers, but we found no way of improving its performance, so we decided to limit its usage. The second function counted the number of alive neighbours of a cell, and in doing so used the aforementioned modulus function and a *for* loop. We decided to unroll the *for* loop and add the values of the neighbouring cells instead of using *if* statements to check if they are alive. After testing both implementations 10 times, we observed an increase in performance of 1.79 times.

3.3 Comparing to the baseline

Our final parallel implementation completes the benchmark in 24.94 seconds on average. The mean relative performance when comparing to the baseline 10 times is shown in Table 2. Although much faster, we concluded that our implementation scales worse than the baseline, as the relative performance decreases with more workers.

Table 2: Comparing with baseline

Benchmark	Relative performance
128x128x2	402.7%
128x128x4	352,4%
128x128x8	322.5%

3.4 Potential improvements

Our Game of Life implementation has the potential to be improved even more. One improvement could be sending the x and y coordinates of alive cells in the halo rows between workers, instead of sending each byte. This should improve performance when boards are not very dense with alive cells. Another obvious improvement to performance would be using more workers (threads), but this is limited by hardware, as a CPU rarely has over 64 threads. We implemented a multi-server version of the simulation, which sends data over the network.

4 Networking

The idea behind the multi-server version of our implementation is simple. We created a new client program that contains just the worker and connection managing logic, and modified our original program to be the main distributor server that controls the clients.

4.1 Multi-server functionality and Design

The server program can handle any number of client instances. These client instances can also have any number of worker instances running on them. For example, if we have 5 servers, one of them is going to host the server program and the other 4 are going to host client programs, each of which can have any number of workers running on them. The client programs connect to the distributor server program, which gives them the initial board sections and necessary information to connect to each other, which is needed in order for halos to be exchanged. User interaction is still functional in this implementation.

4.2 Linking channels with TCP sockets

Continuing with the 4 client example, let's assume that each client is assigned 8 workers by the distributor server. Internally, the workers can exchange halos between them in the same way as in the local version of Game of Life, over byte channels. There will be, however, one or two workers running on this client that each need to exchange two halos to workers on another client instance. This is done by essentially replacing the halo channels with TCP sockets, and sending and receiving data to and from the required clients over the network. In order to avoid sending the halo data through the distributor server, which would result in worse performance, each client listens to a TCP socket, waiting for two other clients to connect to it.

4.3 Benchmarking

By adding multi-server support over networking, we have created additional overhead. We compared our multi-server implementation to the local implementation, and observed that the networking overhead degrades performance (Figure 2). All tests were executed 5 times. The mean of the results was used. We tested the local implementation on a 96 threaded server. The multi-server version used one distributor server with 16 threads and 16 client servers with 8 threads each. These servers were compute-optimised AWS instances.

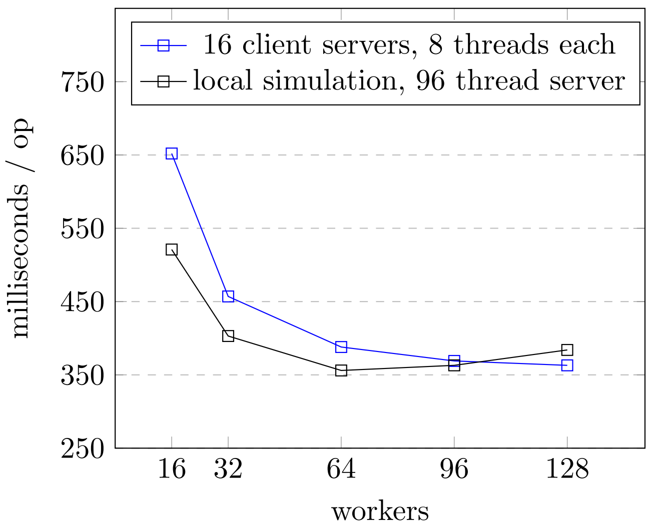

Figure 2: multi-server and local comparison

We concluded that the performance difference between the two versions decreases as the number of workers increases. The multi-server version outperforms the local version when using 128 workers, because the server that the local version was tested on only had 96 threads, the maximum available for compute optimized EC2 instances. This is also why the performance difference decreases as the worker number gets bigger, since there are other *goroutines* besides the workers and even other programs running on the server, so our local version gets slower.

5 Conclusion

We have implemented and analysed different versions of the Game of Life simulation, with different levels of parallelism, and even with networking capabilities. By identifying bottlenecks and thinking of inventive ways of further improving performance, we obtained a local version that performs more than 3.2 times faster than the baseline, on average.
The Game of Life simulation is infinitely scalable, but the number of threads CPUs have is limited, so we created a multi-server version that can use any number of CPUs and tested it on AWS instances.