CSA Coursework: GameOfLife
COMS20008

Valentin Oltyan — xz19539
Armand Cismaru — fz19792

December 10, 2020

1 Parallel implementation

1.1 Functionality & Design

The functionality of the implementation is split between `distributor.go, io.go, gol.go` and `event.go`. The core of the functional design is build around the concept of parallelization.

Considering the implementation requirements, the steps in achieving a stable parallel GameOfLife instance can be broken into the following flow steps:

1. Input is being parsed by `main.go`, consisting of image dimensions, number of threads and turn to be executed.
2. `gol.go` takes the input, initializes channels and calls `distributor` and `startIO` goroutines.
3. The distributor divides the work between workers evolving the board and interacts with other goroutines by sending `Events` back to `IO` using channels.
4. The game is visualised using SDL and can be controlled by user input through a GUI, the distributor also processing the keyboard controls: `'p'` (pause), `'q'` (terminate the program and output the board as a PGM image), `'s'` (send a PGM image with the current board), while reporting back state changes.
5. After all turns have executed the distributor exits the program and outputs the state of the board as a PGM image, regardless of any input keys.

Goroutines paradigm This is achieved by splitting up the input image into (almost) equal chunks and sending them to worker threads (i.e. goroutines), which will then send back the updated state of the board. In the `distributor` method, after input is received from the `ioChannels.input` channel and the first `CellFlipped` events are sent, the image is split into chunks with width equal to *div = ImageWidth / p.Threads*.

For each of the turns, each worker goroutine receives the image size, coordinates for the width they have to operate on, the current board, a status channel and an output 2D slice channel. To avoid race conditions and memory violations, each worker modifies a copy of the board and outputs to an element of an 2D slices array. The goroutines run independently from each other, but the all notify the `done` channel when finishing their execution. To cover the possibility that the board doesn't divide equally given the number of threads, the last call includes *mod = ImageWidth % p.Threads*.

The execution of the program is blocked until every worker notifies that it has finished its execution to ensure synchronization. Following, the board is reconstructed from the output 2D slice channels then to required events are being sent.

Design The distributor is designed to be modular, the functionality is split between the main `distributor` method and the `calculateNextState, calculateAliveCells, calculateNeighbours, makePGM and worker`. The key capturing logic as well as the ticker are implemented in the main method, being possible to be encapsulated as methods but at the cost of redundant code.

1.2 Critical Analysis

Essentially, the efficiency of the implementation is strictly tied to the image size, number of turns and grows with the number threads used. The workers run conccurently, only speeding up the time required for a single update of the board.

The general intuition is that the more workers, the faster the execution. However conceptually true, this is subject to hardware limitations, meaning that from a certain number of worker threads, the efficiency stalls, and from an even bigger number the hardware fails into executing the program. Let's see how and why:

Hardware conditions The implementations scales with the number of threads, because essentially there are more processes that calculate the given board. But, the maximum number of threads that speed up the efficiency is equal to the number of processor threads that the running hardware has. The majority of computers have quad-core processors with 8 processor threads, therefore so many worker threads can work efficiently.

Benchmark When the number of worker threads exceeds hardware limitations, the efficiency becomes flat, as they are not executed conccurently anymore, as one processor thread has to execute parts of more than one worker thread *(Fig.1)*. As the number of worker threads increases, the efficiency still stalls but it can lead to hardware failures when the number becomes too big (e.g. >50).

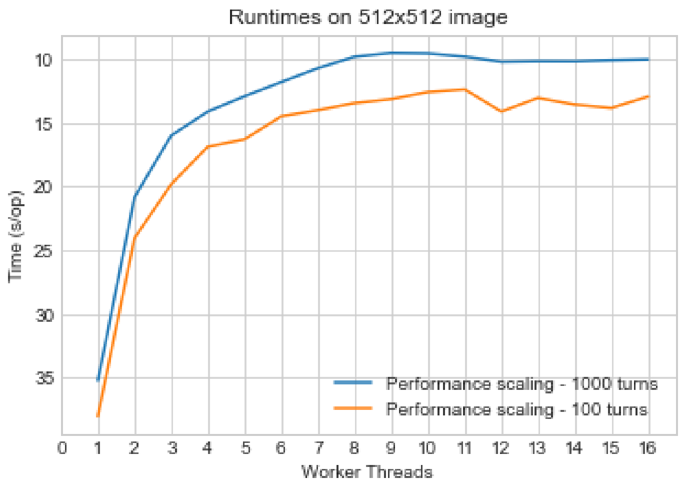


Figure 1: Benchmarks on Linux lab machines, using 12 processor-threads

To properly visualise how the implementation scales, a 512x512 image was chosen, running 100 and 1000 turns. On a smaller image, it would have been more of a test of input reading speed, as the speeds are too fast to gather relevant, noise-free data. When running 1000 turns, the accuracy of the data increases, ruling out even more noise, therefore gathering cleaner results.

2 Distributed implementation

The challenge to design a GameOfLife distributed implementation was tackled using a holistic approach. The components of the system run independently but are interdependent of the others, being possible to add new components (i.e. AWS worker Node) to scale up the implementation. The core flow of the program is similar to the parallel approach but the main difference is that the computation is split into two big categories, the remote (`AWS`) Engine and Worker instances and the Controller client with IO and SDL (`Local Machine`) *(Fig.2)*.

2.1 Functionality & Design

RPC Paradigm On macro level, the functionality is designed keeping in mind the Remote Procedure Call general paradigm. The remote procedure names as well as the structs and types are defined in `works.go`. Every component is initialized through `main.go`, using flags to define the type of component, addresses and instructions such as live visualisation request or resuming board progress in case of the controller reconnecting. Once every component starts, they publish their methods in the `DefaultServer`. When workers are added, they send register requests to `Engine.go` with their IP address and port thus connecting them to the server. With the engine and workers set up and listening for requests, the controller is required to establish connection to the remote server in order to start the simulation.

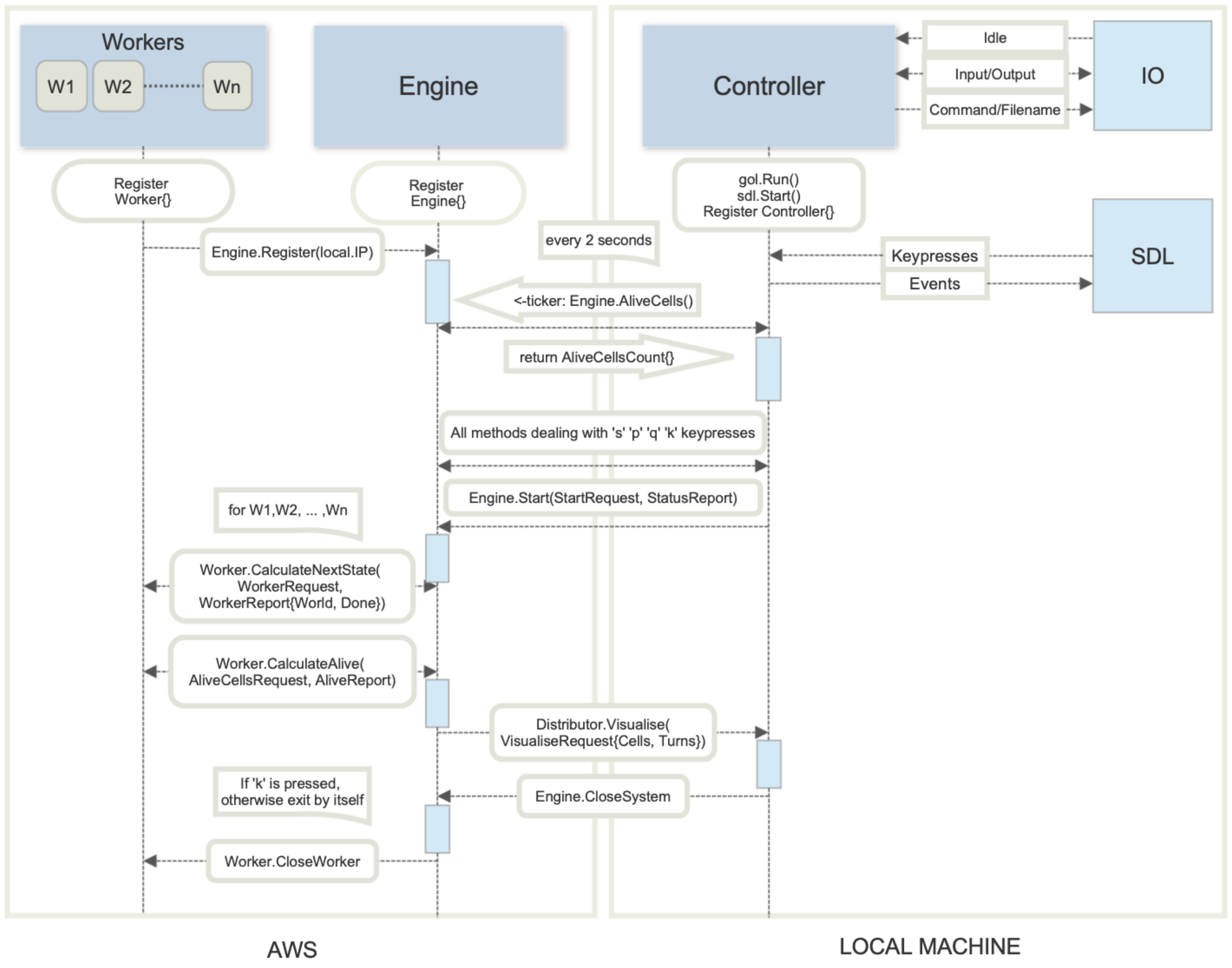

Workers
W1
W2
Wn
Engine
Controller
Idle
Input/Output
Command/Filename
IO
Register Worker{}
Register Engine{}
gol.Run() sdl.Start() Register Controller{}
Engine.Register(local.IP)
every 2 seconds
Keypresses
Events
SDL
<-ticker: Engine.AliveCells()
return AliveCellsCount{}
All methods dealing with 's' 'p' 'q' 'k' keypresses
Engine.Start(StartRequest, StatusReport)
for W1,W2, ... ,Wn
Worker.CalculateNextState(WorkerRequest, WorkerReport{World, Done})
Worker.CalculateAlive(AliveCellsRequest, AliveReport)
Distributor.Visualise(VisualiseRequest{Cells, Turns})
If 'k' is pressed, otherwise exit by itself
Engine.CloseSystem
Worker.CloseWorker
AWS
LOCAL MACHINE

Figure 2: Conceptual macro-diagram of the Distributed implementation

Flow & Task approach The flow steps below are conceptually illustrated in the diagram above, with regards to the logic flow of the running instances.

1. The starting point of the simulation is the controller which, similarly to the parallel implementation, takes the input image from IO and sends back events to SDL.
2. The controller calls the key processing logic goroutine, waiting for and processing key controls until the simulation ends or being told to do so. Depending on the pressed key, the controller can request to Pause, Unpause the simulation, output a PGM snapshot of the current BoardState, disconnect the controller or stop the simulation and cleanly close the system.
3. Every two seconds, the `RequestAliveCount` goroutine requests the number of alive cells from the Engine and sends `AliveCellsReport` events back to SDL.
4. While the aforementioned operations run concurrently, the simulation only starts when `Engine.Start` is called with the required parameters. It sends back a `StatusReport`with the number of completed turns and the final board.
   (a) In order to keep the network transfer to a minimum, the engine would split the board into smaller slices when communicating between the engine and its workers, sending only the necessary amount of information for evolving that slice.
   (b) The coordinates for the image sliced, as well as the worker number are being passed through the `startWorker` goroutine that sends a `Worker.CalculateNextState` request over the network, getting back the updated board and notification of the work being done.
   (c) To ensure effectiveness in updating the board, the worker sends the image slice and its neighbouring left and right rows to the `calculateNextState` method, returning an updated version of the sliced board.
   (d) The implementation offers the possibility to use worker threads. But, a problem encountered was how to split the board for the multiple threads on each worker. Knowing that the board was already split once on x, the best solution was to split it on y, creating a more even division of work.

Live visualisation extension The system also offers the possibility to get back the visualisation from the board evolving through the remote workers, the engine requesting the AliveCells back from the worker, sending them to the controller only if the engine instructed to do so when first started. To note that this option has a significant impact on performance, being disabled by default, only enabled when the flag `-Visualise=true`, along with the requirement of opening a new listener on a new port which may require port forwarding when working from a personal computer. When designing the visualisation of the board on SDL, communication problems aroused, requiring to create a listener on the controller and establishing a new data transfer that would happen every turn.

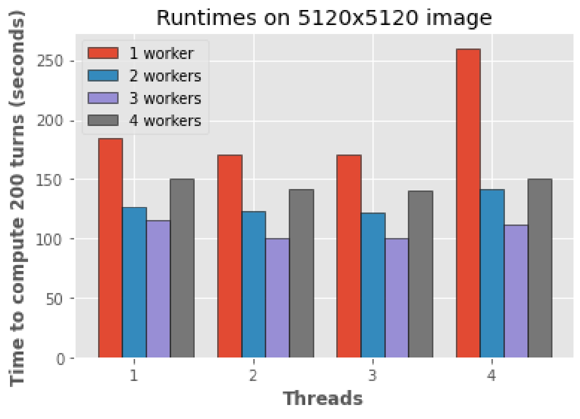
Fault toleration In the case of a worker disconnecting, the engine will throw a `panic` signal telling that it has lost connection to the respective worker. If the engine disconnects, the controller will throw an error and the workers would remain idle. And, lastly, if a controller disconnects, the simulation will continue to run normally, behaving the same as in the case of a `'q'` key press. The program is designed in such a way that if the controller restarts, it is possible to reconnect to the engine and see what the current state of the board is.

2.2 Critical Analysis

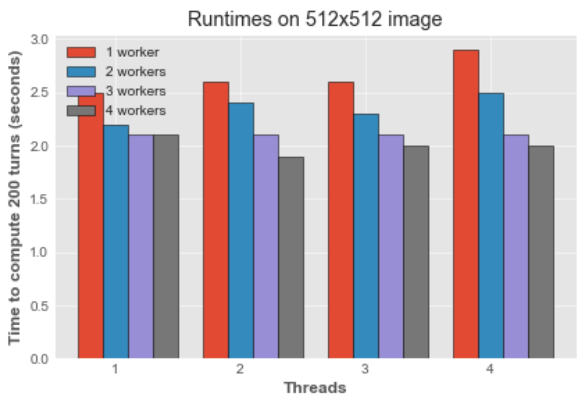
Fig. 3 below represents the runtimes of two different board sizes executing 100 and 200 turns with different configurations of workers and the number of threads each worker can use. The AWS instances used for this benchmark are `c4.xlarge` for both the engine and the workers, while the controller runs on the lab machines. Outputting PGM files after all turns are have completed is also deactivated to prevent noise created by the distributor, although reading still affects the runtimes.

Benchmarking using threading Important to note is that the `c4.xlarge` AWS instance has 4 available virtual CPUs, meaning that using over 4 threads per worker would negatively impact the performance. The same thing applies to the engine, because for the RPC call of each worker we use a different goroutine so having more than 4 workers would impact the performance negatively. For a different engine configuration, more workers could be added.

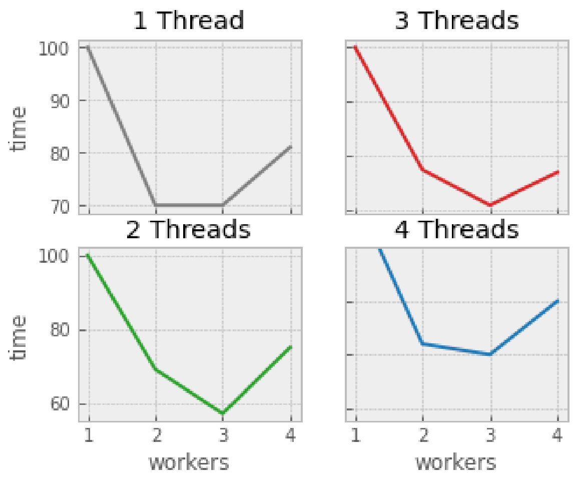
Threads x Workers performance From the barplots and graphs in Fig. 3 it can be observed that, in general, adding more workers and threads speeds up the calculations of the board, the best performance being achieved by the 3 workers, each with 3 threads, being almost twice as fast as one worker with one thread when calculating 200 turns of the 5120x5120 board. One thing that stands apart is the fact that adding a 4th thread starts slowing down the runtime. This happens because of the way the workers function operates: for each thread, the worker needs to create a new goroutine, however, there is already a goroutine that serves as a listener and the function that calls the goroutines itself, this resulting in more functions than the worker can do in parallel at the same time. The same is also true with adding a 4th worker to the number of workers for the engine. More workers result in more goroutines that need to be processed by the engine at the same time as the listener and the function that calls all the goroutines, resulting in a decrease in performance.



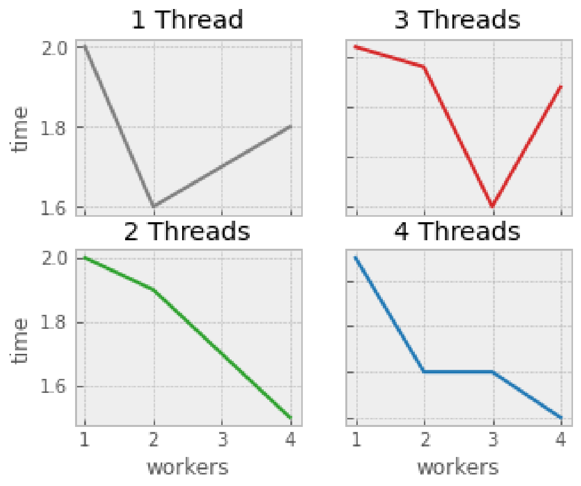
(a) 5120x5120 - 200 turns



(b) 512x512 - 200 turns



(c) 5120x5120 - 100 turns



(d) 512x512 - 100 turns

Figure 3: Benchmarks on different picture sizes and turns.

Note: The two types of data plotting were chosen for diversity. On the three threads graphs, the lowest point is actually lower than represented, due to Matplotlib formatting.