Computer Systems A Report – Game of Life

Rares Bucur, Valentin Papa

1. Introduction

This coursework consisted in taking various principles from concurrent and distributed computing and implement them into the Game of Life system. Throughout this assignment we tested each of the stages looking for different speeds of processing in multiple scenarios. We also benchmarked the final version of our program and then compare the results to the base case which held just the logic without parallelism.

2. Functionality

2.1 Serial Implementation

In the first stage we had to create a simple implementation of Game of Life, without using the principles of parallelism. We successfully created a serial implementation of the game's logic, utilizing only a single processing thread and calling the ''*nextIteration*'' function just once every turn. Our implementation involved reading in an input board state to a 2D slice of bytes. Then, every turn would update the world state into a slice this way.

2.2 Parallel Implementation

During the second stage, the principles of concurrent programming were implemented into the system. The task of processing each turn was to be distributed between multiple workers/threads, each one of them being responsible for an equally sized board segment, but we thought about how to break the problem down into non even segments. The requirement was to support all multiples of two, however we found a solution which works for any number of worker threads. For this to work, we had to figure out a standard thread size and then giving the final thread take the rest of the board.

2.3 User input & Visualization

After we had implemented the parallelized version, we made use of the "*AliveCellsCount*" event to see how many cells are left alive after each processed turn. We also used a ticker goroutine to report the number of cells still alive every 2 seconds.

We implemented an output logic which returns the state of the board after a turn is completed. For that to happen, we use the "*activateWritePgm*" function which is activated by receiving 0 on the channel and the path of the file that needs to be and written. We then store the output into a "*finalWorld*" matrix which is sending the bytes to the "*outputChannel*".

For the last stage, we added 3 keypresses that the user could use during runtime. This involved making a "*manageSdl*" function which changes the value of "*operationType*", because for example, we cannot stop the process while copying the world; and making a "*checkOperation*" function which makes sure if an operation can be done after a turn is processed. When the key "s" is pressed, it exports the current state of the board to a PGM image and when the "p", "q" keys are pressed, it pauses or quits the program.

3. Distributed Implementation

3.1 One node implementation

For the distributed system, the code from the parallel system stayed almost the same but in addition, we made a separate "*server.go*" file for the AWS nodes part and integrated it within an RPC system that makes the connection between our controller and the servers. For the first part, we started by implementing a basic controller which can tell the logic engine to evolve Game of Life for the number of turns specified.

3.2 Multiple nodes implementation

For the second half of the implementation, in comparison to the parallel implementation where every thread is allocated to work on just a segment of the image, now, alongside each piece of the image we send to the workers to process, we will send the above and below row of that piece as well to work more efficiently. They would then perform the logic and reconstruct their part of the board. This is done with the help of ''*executeThread*'' function and ''*worldSlice*'' which takes the ''*lowerIndex*'' with ''*upperIndex*'' and append them to the part of the image which is being processed and requested by the controller. After that, we call the function ''getNextIteration'' which puts in the response the next state of that part of the image which has been sent.

The last step was to alter our keypress logic so that when the ''k'' key is pressed, all servers alongside with the client are shut down and the system outputs a PGM image of the final state. That was made possible by calling the function ''StopServer'' from the server in which if the requested message is ''k'' then it exists the process.

A live viewing of the distributed game of life was also implemented which lets the user see each turn being processed live and interact with the system saving the current turn, pause or quit the program. As a side note, if let us say, a component of our system may disappear, for example a malfunctioning server, that would affect the system because if the server will be back online, the program would resume.

4. Benchmarks

The following benchmarks show the results for the single-threaded implementation in comparison with the multi-threaded one for the parallel system as well for the distributed part for one AWS node vs multiple nodes. The tests were run on one of our personal local machines. To be assured that the benchmarks reflect a result as good as possible, no other programs were running in the background during the execution.

Table 1: Serial benchmark results

Output	Benchmark Time(s)
16x16x0-1	0.01
16x16x1-1	0.01
16x16x100-1	0.01
64x64x0-1	0.03
64x64x1-1	0.03
64x64x100-1	0.9
512x512x0-1	1.05
512x512x1-1	1.09
512x512x100-1	4.93
TestGol	7.278

Table 2: Parallel benchmark results

Output	Benchmark Time(s)
16x16x0-16	0.02
16x16x1-16	0.01
16x16x100-16	0.01
64x64x0-16	0.03
64x64x1-16	0.03
64x64x100-16	0.08
512x512x0-16	1.05
512x512x1-16	1.07
512x512x100-16	3.46
TestGol	5.787

Table 3: Distributed benchmark results with one AWS instance

Output	Benchmark Time(s)
16x16x0-1	0.15
16x16x1-1	0.29
16x16x100-1	13.51
64x64x0-1	0.17
64x64x1-1	0.31
64x64x100-1	13.73
512x512x0-1	1.83
512x512x1-1	3.18
512x512x100-1	34.43
TestGol	67.696

Table 4: Distributed benchmark results with 8 AWS nodes

Output	Benchmark Time(s)
16x16x0-8	1.07
16x16x1-8	1.22
16x16x100-8	14.79
64x64x0-8	1.10
64x64x1-8	1.22
64x64x100-8	15
512x512x0-8	2.75
512x512x1-8	3.14
512x512x100-8	18.63
TestGol	59.031

5. Conclusion

This assignment has thought us how large of a difference parallel and distributed implementation can have on the computing speed of a program. Efficient use of the worker threads can speed up the process to even around 2x of execution for bigger images when compared to the initial one threaded implementation.