COMS-20008 Computer Systems A - Game of Life

Sebastian Whipps **nk22674** & Valentina Velasco **xd22898**

Contents

1 Parallel

1.1 Implementation

Our parallel implementation of John Horton Conway's Game of Life aims to run reliably and efficiently. We wanted to implement a solution that scales well regardless of the given world size and threads. Our implementation passes all of the tests and the key-presses function as required allowing the client to output the current state, pause or terminate the program.

1.1.1 Establishing Channels

In `gol.go`, `Run()` establishes channels to allow for communication between the `io` and the `distributor`. `startIo()` and `distributor()` are both called as go-routines, and run concurrently. They both take `p` as a parameter, which contains the world dimensions, the number of turns and the number of worker threads. They are then given their shared channels. `distributor()` also takes the `events` and `keyPresses` channels enabling communication with the SDL logic.

1.1.2 The Distributor

The distributor executes the turns of the Game of Life. Each turn involves dividing the work among multiple workers to process different sections of the game world concurrently. Initially, it loads the world from the input channels, populating the `[]util.BitArray` that represents the game grid.

As it progresses through the turns, the distributor divides the work amongst the `worker()` go-routines through `threadScale()`. These workers run concurrently each turn, with each worker responsible for a part of the world. Once the workers complete their computations, they send back

their processed segments to the distributor via a unique channel. The distributor then assembles these parts into a new world state, and updates the main world slice accordingly.

The distributor also communicates with other go-routines each turn, to handle key-presses to potentially pause or halt the Game of Life simulation. There is also a `Timer` that communicates the current turn and the number of alive cells every two seconds via the events channel.

The distributor continually loops through turns until it reaches the specified number of turns or receives a halt signal due to user input. Upon completion or halting, it initiates `exit()`, this saves the world in the current state and ensures that the program terminates gracefully.

1.1.3 Scaling Threads

The `threadScale()` function allocates threads for an efficient distribution of the computational load, enhancing the overall performance of the Game Of Life. It pre-defines the amount of work each worker should undertake based on the height of the world and the number of available threads. This is done by calculating the base workload for each thread and considering any remaining workload after the base distribution, this ensures an even distribution of tasks. This load balancing prevents any single worker from being overloaded with excessive computations. The resulting `scale`, is a slice indicating the workload for each thread.

1.1.4 The Workers

To concurrently process the Game of Life board, we implemented a worker function. Initially, we passed the entire world into each worker. However, we realized it's only necessary to pass the part to be processed, along with the row above and below, optimizing resource usage. To address the world's wrapping around the edges, we created a `TransformY()` function, ensuring proper indexing by handling negative values and those exceeding the height.

During each turn, the workers are invoked with a specific number of rows to process. Iterating through each cell within the designated rows, they count live neighbors and adjust the output for the next part of the world. Any changes, such as cell flips, are communicated to the main world through an events channel. These worker operations are carried out concurrently using goroutines, enhancing performance.

The next part of the world is returned via a channel, maintaining an efficient and non-blocking communication pattern. This design choice allows for the continuous execution of turns without unnecessary waiting. Overall, the worker implementation optimizes the Game of Life simulation by distributing the workload effectively among concurrent processes, leading to improved efficiency and responsiveness.

A further improvement to this would be to have the workers execute multiple turns at once, and pass the overlapping edges between them between the turns.

1.1.5 The `BitArray` Data Structure

Our initial method of storing the current world state was through a 2d slice of byte values, `[][]byte`; 0 and 255 representing dead or alive cells respectively, with the outer index representing the row of the game and the inner row representing the column. However, this solution was memory inefficient, since there are only two values a cell can take, not 256. Hence we decided to store each cells value as a bit rather than a byte. After some research on how we could implement this while maintaining good read and write speed [1]. We concluded that it was best to create an array of `uint8` (equivalent to `byte`), where each bit of the `uint8` represents the cell value. However, we did need to create a new way of indexing the `BitArray` for getting and setting bits using bit masks.

To represent the world, we used a slice of `BitArrays` to store the data row by row. We did consider creating a wrapper for this called `BitGrid`, but decided it would add unnecessary complexity. A potential improvement would be to use run length encoding, which would significantly reduce

memory usage. This would work well, since the world contains large sequences of dead cells. We decided to leave this as it would complicate the process of read and write operations, as indexing changes as a result of using a variable length structure.

1.2 Benchmarks

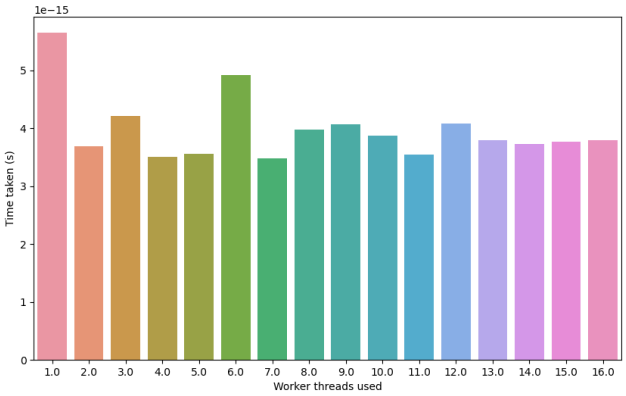

Figure 1: Average time per operation (512x512x1000) over range of threads for 50 trials. Tested on an 8 core M1 Mac

The benchmark results provided shows that as the number of threads increases, the execution time generally decreases, indicating that parallelisation is improving the overall performance. We ran the benchmark multiple times in order to improve the accuracy of the results. This helps in minimizing the impact of external factors and obtaining a more reliable average performance measurement. While adding more threads improves speedup, the fact that the time taken stays constant (lower but constant) after the first thread suggests that the overhead of managing more threads might start to outweigh the benefits gained from parallelisation.

2 Distributed

2.1 Implementation

Our Distributed implementation passes all the given tests and correctly outputs the PGM files when complete or in response to a client-side key-press. All the other key-presses also function as required allowing the client to pause, quit or kill the system. The worker nodes and broker all run on separate AWS nodes, with the client operating on a local machine to allow for SDL to handle the key-presses.

2.1.1 RPC

The first part of our distributed implementation involved taking the serial implementation for our parallel program and separating it into two components. One component, the local controller (client) and the second component, the GOL Engine (the server) containing the game of life logic. Our distributed implementation is based on the Publish-Subscribe model so instead of having channels to receive the new world, we set up RPC connections between the two as the controller/client would need to send the initial world along with the number of turns and image to the server/engine and receive the final world from it. In order to define the methods needed for these RPCs, we created a stubs file, which at this point only contained `RunGameOfLife()`. As we continued with the implementation of the distributed component, in order to report the number of alive cells and

manage the behaviour of the GOL engine through key presses, we defined more RPC connections and proceeded to add more workers so that they would run in parallel to compute the new world. Whilst doing this, in order to reduce coupling between the controller and the GOL workers, we developed `broker.go`. To initiate communication, the Controller connects to the broker (which returns the final game state once it is finished) and the broker connects to the Workers (the broker sends the slices with the adjusted height depending on the number of threads - in our case, 4).

For the 'k' case, designed to gracefully shut down all components of the distributed system, we started by terminating the workers. This process involved establishing an RPC connection between the controller and the broker, prompting the broker to instruct all active workers to shut down. Subsequently, the broker then initiates another RPC connection with each worker, requesting them to close their TCP listeners. Closing these listeners ensures that worker nodes stop accepting new connections and undergo a smooth shutdown. Then, we proceed to shut down the broker using a `boolean` field in the `GameOfLifeOperations struct` that closes the listener - essentially stopping the broker. The final step is shutting down the client which is done through a defer statement which closes the RPC client connection when the `distributor()` function completes its execution.

2.1.2 The Broker

The broker serves as a central coordinating system in the distributed implementation of the Game of Life. It solves the problem of parallelising the simulation by primarily focusing on workload distribution and the communication with worker nodes. The broker efficiently divides the computational load amongst multiple workers, addressing the challenge of processing large game boards by assigning smaller parts of the world to individual workers. This load balancing ensures that each worker is responsible for a manageable portion of the game world, promoting parallelism and efficient resource utilisation. The distributor is the client and it coordinates the overall execution of the simulation. It interacts with the Broker through remote procedure calls to initiate, control, and monitor the Game of Life simulation.

In order to improve the broker, we could have externalised configuration setting such as the server addresses into a configuration file. This makes the code more flexible and easier to configure without modifying the source.

2.1.3 The Workers

We currently have four worker nodes hardcoded into the broker, and this configuration can be adjusted in `stubs.Threads`. The workers operate in a manner similar to the parallel implementation—they await their turn, process their designated portion of the world (which includes the row above and below), and subsequently return the updated value for their specific section. It's worth noting that while the workers run on separate AWS nodes, their code remains identical.

2.1.4 Extensions

Parallel Distributed System In worker.go, we have a function `subDistributor()` that further partitions the world on each AWS node. This function runs multiple go routines on each node, which is currently configured with 4 threads per node (although we do have dynamic scaling to further improve this in the future). Implementing this was relatively straightforward as it built upon the concurrent task. Since all operations were confined to the same node, we could once again use channels to pass the computed parts of the world from the concurrent go routines.

Fault Tolerance When we initially implemented the broker, it became apparent that if we retained the world and completed turns count from the first call of `RunGameOfLife()`, we could continue the processing from where we left off. Therefore, provided the broker has not halted in the meantime and the workers are alive, the program continues. A slight tweak we made to this

was adding a resume `boolean` that was passed into the `RunGameOfLife()` RPC request. This is due to the fact that we do not always want to resume the Game of Life, such as in the case of testing. Therefore the program will not resume when the number of turns is below one million. This means that running main with no arguments will set resume to true, but all the given tests and our benchmarks will not. Perhaps this is not the most ideal way of implementing this, but the alternative of adding parameters to `Run()` in `gol.go` did not seem much more efficient.

A potential improvement to this would be initiate the workers from the broker, so that we can avoid the scenario where the broker is called but has no way of processing the data. Another refinement would be to save a PGM file on the broker whenever connection is lost, so that it can be resumed even if the broker halts mid processing.

2.2 Benchmarks

2.2.1 Local Broker & Workers

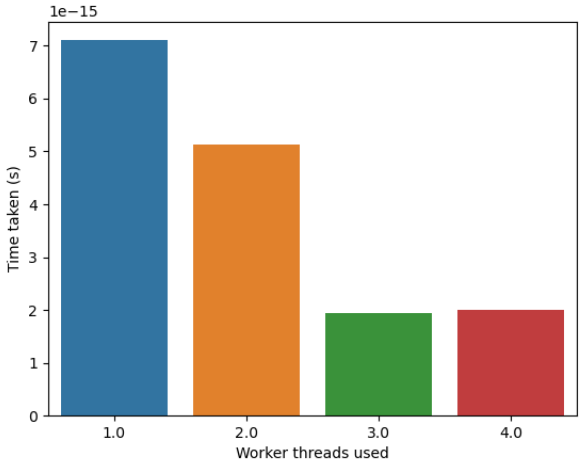

Figure 2: Average time per operation (512x512x1000) over range of threads for 50 trials. Tested on an 8 core M1 Mac

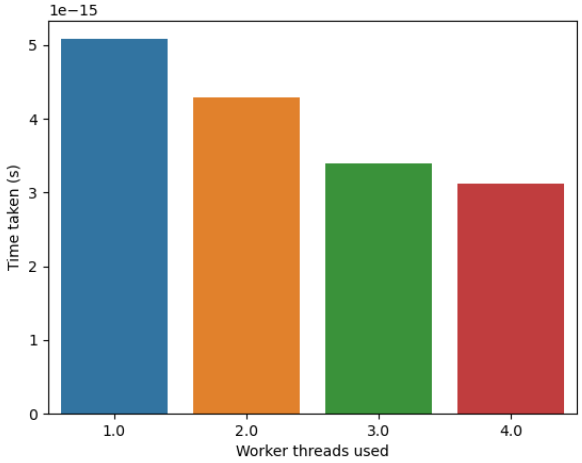

Figure 3: Average time per operation (512x512x1000) over range of threads for 50 trials. Tested on the university's 6 core lab machine

Analysis On both architectures, the benchmark results provided show that the time per operation decreases with an increasing number of threads, which suggests good parallelisation. The Intel x86 architecture demonstrates more efficient parallel scaling with decreasing time as the number of threads increases. In contrast, the Mac exhibits less consistent scaling, with a significant improvement from 2 to 3 threads but diminishing returns from 3 to 4 threads. Variances in hardware architecture, cache structures and inter-processor communication between the Intel x86 system and the Mac can impact parallelised code performance. The specific features of the Intel x86 architecture may contribute to better parallel scaling.

2.2.2 Broker & Worker on AWS

We created 5 separate AWS medium EC2 instances - four running `worker.go` and one running `broker.go`. Subsequently, we initiated the client (`distributor.go`) on our local machine, providing the address of the broker node as an argument. Executing the Game of Life in this distributed manner enabled us to achieve an approximate throughput of 125 turns per second. This performance closely resembled the local runtimes observed on x86 CPUs, however it was slower compared to the local performance on an M1 Mac chip. This suggests that further improvements in runtime could be achieved by delving into how our program maximizes the computational capabilities of each node. We could also optimise our worker code to align with the architecture of AWS nodes and make some enhancements to the overall architecture of our distributed system, which may contribute to a more efficient execution.

Executing only the workers on AWS nodes while keeping the broker local resulted in significantly slower performance compared to the previous setup. This outcome is expected, considering that a substantial portion of RPC requests and responses occur between the broker and the workers. Therefore, the higher latency between the local machine and the AWS nodes is certainly a limiting factor. Notably, the communication between AWS nodes have low latency, given that the nodes are situated within the same region.

To improve the implementation further, we could explore reducing the number of RPC requests by having the workers execute multiple turns. This can be done either by inter-worker communication, or having the workers process overlapping parts and return only the precise middle rows of each part.

References

[1] Deleplace, Val. "7 ways to implement a Bit set in Go - Val Deleplace - Medium." Medium, 27 Feb. 2023, https://medium.com/@val_deleplace/7-ways-to-implement-a-bit-set-in-go-91650229b386.