Report

Euan Bagguley, Joshua Hall, Samani Mukhtar

11 December 2020

Part I
Parallel

Functionality

We successfully pass all test cases (`TestGol`, `TestAlive` and `TestPgm`) for this section. Our implementation uses the correct number of workers defined in `gol.Params`. It also correctly displays the live progress of the game via SDL and interactions with the system via keyboard controls work properly. Our implementation is free of deadlocks and race conditions.

Communications Between Goroutines

The first instances of goroutines we use are when the `Run` function in `main.go` triggers the distributor and `io` files to run. Within the distributor goroutine we trigger the reading of a file by sending a read command down the `ioCommand` channel, it's received by the `startIo` goroutine which runs the `readPGM` function. This function waits to receive from the `filename` channel, meanwhile the distributor has progressed so it is now ready to send the name of the file down the `filename` channel. The `startIo` goroutine then reaches a point where it is ready to send the image it has translated to a slice of bytes down the `input` channel byte by byte. The distributor goroutine receives these bytes one by one and populates its world slice. Interaction between the two occurs again when outputting a PGM image. This interaction is similar to reading an image: a command to start running `writePgmImage` function is sent from distributor to `startIo`, `startIo` waits for a filename which is sent by distributor down the `filename` channel, then populates a world by receiving each byte, cell by cell, from the distributor goroutine on the `output` channel. The final time interactions occur is when distributor is checking if it is okay the quit, the `startIo` goroutine will reply and if it is idle the program is quit.

The only other goroutines we use are for the workers. These are triggered to run by the distributor and a new worker goroutine is run for each thread. When these have completed their processing they send their respective worlds byte by byte back down their own `workerOut` channels to the distributor. This goroutine receives these values showing there is interaction between the two goroutines.

Implementation Design

We decided to split the world into blocks of rows, with the number of rows in each block being determined by `ImageHeight` divided by the number of threads specified in `gol.Params.Threads`. Upon running the tests we discovered this method only correctly split the world when the `ImageHeight` was divisible by the number of threads, we now had to account for remainders (leftover rows were not assigned to any section in the now split up world).

The first solution we found for this was to calculate the remainder via `ImageHeight` modulo `gol.Params.Threads` and increase the size of the final block of rows by this remainder. Although this worked successfully we saw this may not be a very efficient solution. For example, if the `ImageHeight` was 64 and the `gol.Params.Threads` was 13 then the split would go as follows: blocks 1 through 12 containing 4 rows and block 13 containing 16 rows (4 + (64 modulo 13)).

Our final solution to splitting the world, in theory, was more efficient. We altered our program so that if the remainder is greater than one then the goroutine worker we create has a block size of (`ImageHeight / gol.Params.Threads`) + 1. We then decreased the remainder by one each time this happened. Thus a split of 64 height and 13 threads now looks like: blocks 1 through 12 have size 5 and block 13 has size 4. This shares the remainder a lot more evenly and doesn't end up with a final worker being much larger than all the others.

We chose to split the world up into parts and only send a single part to each worker goroutine to try and maximise efficiency. The other method we considered was to send the entire world into each worker and specify which of the rows in this world each goroutine should execute a turn on. Our method of implementation is more efficient as it sends less data to each goroutine meaning we use less memory and perform fewer communications.

Critical Analysis

Figure 1: Parallel implementation - Comparison of the benchmark results we gathered from our two different attempts

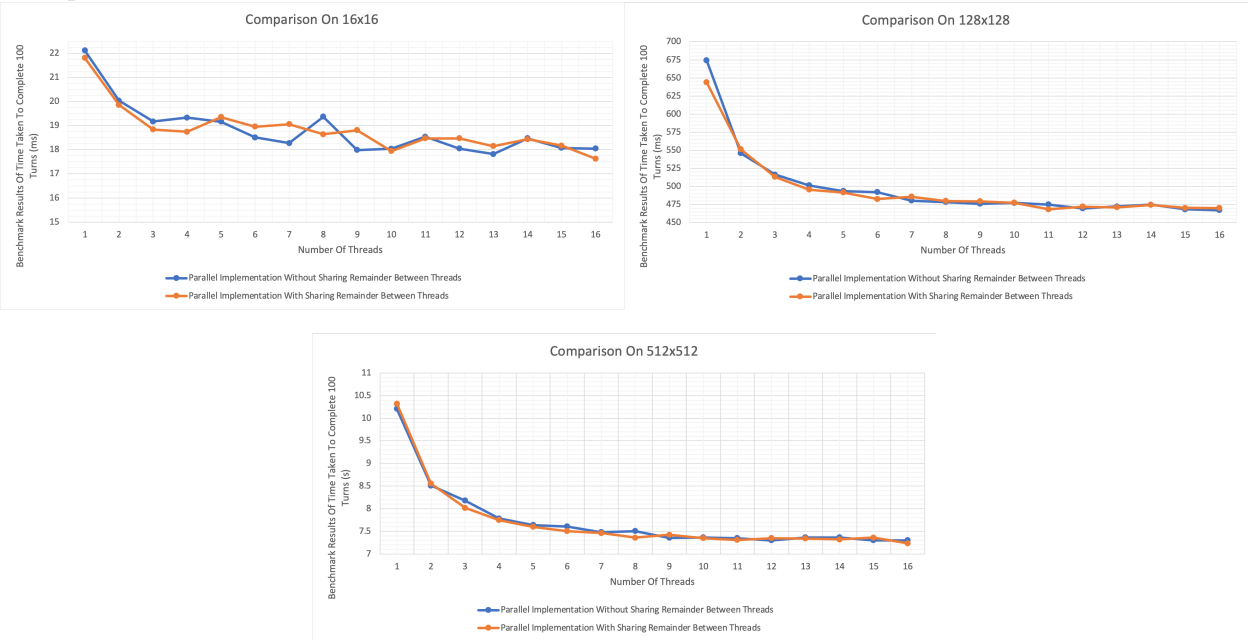

Figure 1 shows the graphs we generated using the benchmark times gathered from running 100 turns of the game on respective image sizes when increasing the number of threads for both our first and final implementations of the parallel system. It clearly shows that there is negligible difference in the two different implementations. We believe this is because both

implementations must be bottle-necked by communications down channels slowing them down as opposed to waiting on workers to complete their processing. A solution to reducing communications may be to edit the worker functions so that they send back their block of the world after completing all turns instead of only doing one turn at a time.

Before performing the benchmark tests we envisioned that the greater the number of threads, the faster the execution would be. We thought this as splitting the work between multiple workers means that each worker has a smaller amount of cells to process and each could run in parallel. Our benchmark results support this. As shown in the graphs, when the number of threads and thus workers increases, the time to complete 100 turns decreases.

We notice that in all the graphs in figure 1 the time to complete 100 turns decreases rapidly at the start and at some point starts to plateau. We expected that dividing the work between multiple threads against the time it takes to process 100 turns having an exponential decline, however, the plateau occurs sooner than we had thought. We believe this is due to communications between the workers and distributor using channels slowing down the process at run time. Here the efficiency of the system becomes more dependent on limiting communication than splitting work.

We also notice the number of threads at which the plateau begins seems to increase as the size of the image increases. This is because the impact of dividing the work between a greater number of workers on a larger image still outweighs the effect of increased communications at a higher number of threads. For example for a 16x16 image. When the threads increase from 4 to 5 the number of rows in each split block only goes from 4 to 3 (1 thread has 4 rows). However, when the number of threads goes from 4 to 5 in a 512x512 image the block sizes go from 128 to 102. Decreasing each block size by 26 is a lot more tactful on reducing run time than reducing each block size by 1.

Part II
Distributed

Functionality

Our system covers almost all the success criteria. We successfully past all test cases (`TestGol`, `TestAlive`, `TestPGM`). We have successfully implemented all functionality required for the original keyboard presses (s, p, q). When using the system you can successfully connect and disconnect new controllers, ensuring when a new controller connects it either takes over the control of the current game or starts a new one when necessary. It outputs the correct PGM images. Finally it is able to run with multiple AWS nodes. The only functionality we are lacking is the final output image being produced when k is pressed, however, we do gracefully shutdown of all components when pressing k.

Implementation Design

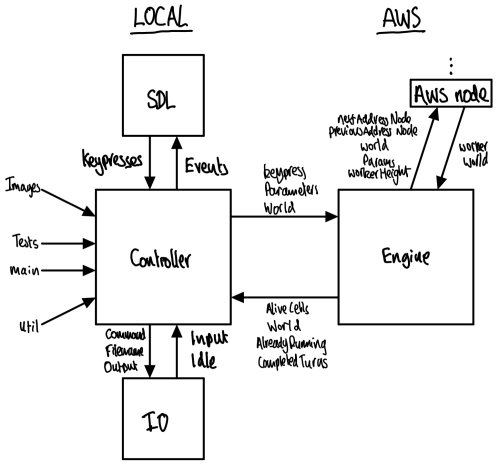

Figure 2: System design

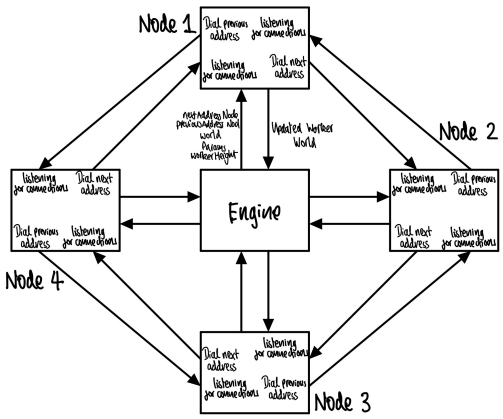

Figure 3: Sync design

Figure 2 shows how we decided to set up our distributed system. The first big decision we made was where to run the turns loop, in the controller or the engine. We decided to put it in the engine so that that communication was reduced (otherwise the world would have to be sent to the engine and received back from the engine every turn). This turned out to be the correct way as if turns loop was in the controller the engine wouldn't be able to continue executing once the controller disconnected.

Figure 3 shows our next big design feature: how the nodes will communicate efficiently. To ensure there was minimal communication we used a halo exchange scheme. As shown in the diagram we implemented this by having the engine act as a "master" node. It passes to each node: its respective section of the world to run an iteration of the game on, the `params`, the height of the section and the addresses of the previous and next nodes. For halo exchange

the key factors sent here are the addresses of surrounding nodes. The system works by first sending the required data to all the nodes, next the addresses are sent to all nodes and each node connects to its surrounding nodes to form a cycle. Once the cycle is formed all nodes wait to receive the trigger from the engine to start execution. The nodes get their next row from the next node and their previous row from the previous node to form the "halo" around their section.

In our original implementation the connection between the engine and the nodes was severed and reconnected multiple time causing the program to run extremely slowly and often exiting with a too many files open error. We solved this by finding a way to maintain a single connection throughout the whole execution.

We could further improve our design is by having the engine send off the split up initial world to each node and where they operate all turns until either a request from the engine tells them to return the updated world (due to key press or failure) or they complete all the turns, then return the world.

Data Sent Over the Network

After establishing the connection between the local controller and the AWS engine the first function the controller calls is the `isAlreadyRunning` function on the engine. This sends the controller `Params` to the engine and waits for a boolean reply. We send this to establish if the engine is already running and if it is, does this new controller have the same parameters? If this function returns true this controller takes control of the already running world, if it returns false the engine ends the current running world, if there is one, and starts a new game.

At this point, there are two paths the controller may take. In one path the next transfer of data that can occur is done through the controller calling the engines `Start` function. Here the controller passes a request containing a world (read in from an image specified by its `Params`) and its `Params` across the network to the engine. This is done if either: there is no game running on the engine or if the controller is not compatible with the currently running game. It needs to pass these to the engine so it can start the running of the game.

If the controller does not take this path then it instead calls the engines `Continue` function. Here the controller passes "0" to the engine, this is just a place holder as the engine does not require anything to be passed to it as the controller will just assume control of the current game.

Whichever of these two paths the controller takes both wait to receive a world back from the engine. Both of these only return the world when the engine finishes executing. This occurs if the turn limit imposed on the engine is reached or if the current game is ended to start a new one. The world is received back for three reasons: to calculate the `finalALiveCells`, to send a `FinalTurnComplete` event to inform the testing framework of the updated state of the world, to create a PGM image file of the final world.

The controller and engine attributes of the network also communicate every time the ticker ticks. Thus every two seconds the controller calls the engines `GetAliveCells` function. This sends a placeholder "0" to the engine, triggers the function to run and waits for the response. `GetAliveCells` replies with a structure containing the current number of alive cells and current turn. These values are then used to send an `AliveCellsCount` event, outputting the completed turns and number of alive cells on the controller as required.

The only other times the controller and the engine communicate is when a key press occurs. All key presses(s, p, q) cause the controller to send a placeholder value as a request to the engine. Then each causes the run of their respective functions in the engine. The replies received are are all custom structures containing the required information to send their specific events down the events channel and to the SDL.

Critical Analysis

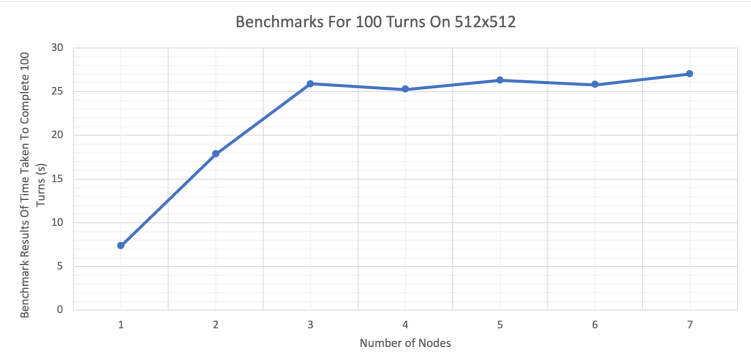

Figure 4: Distributed Implementation - Benchmark results

Figure 4 shows that as the number of nodes increases, so does our run time. As mentioned previously our design is not as efficient as we would like. On each turn we split the world, send the splits off to their worker nodes, regain the updated splits, and then rebuild the world. If we had more time we would like to implement the improved design we mentioned in the design section of this report.

Fault Tolerance

If our system loses a component for any reason during execution the system fails. This is because lots of components are reliant on each other, for example the engine regularly calls nodes, nodes call each other every turn and even the rebuilding of the world is dependent on all components being present.

Methodology for acquiring results and measurements

We used the same benchmark to test the run times of both our distributed and parallel implementations. The benchmark creates an array of `gol.Params` and then loops over them, testing the run time of `gol.Run` for 100 turns for each set of dimensions.