CSA Coursework: Game of Life

Marcus Abraham (dn19300) & Martha Baylis (tj19472)

Functionality and Design

Our core Game of Life logic is implemented so that it runs correctly for a single-threaded implementation, which can then be used in our further developments of other approaches.

Parallel Implementation

Our parallel implementation revolves around the `distributor` function, which creates a slice to store the world, populates it from an input, and identifies the cells that are initially alive. It then identifies the number of threads required, as given by the parameters handed into the program when it runs. It then splits up the world into these threads.

This process occurs in the `splitWorld` function, which takes the parameters, the world, and the width of the world divided by the number of threads - the `threadWidth`. Because the width of the world is not guaranteed to divide evenly among the given number of threads, it is necessary to make sure that for cases where the calculated `threadWidth` is a decimal, the program can correctly identify this and compensate for it - rounding down the width of each slice and add the remainder to the length of one of the slices.

The `splitWorld` function uses `threadWidth` to slice the world into a number of slices equal to the thread number, which it then returns in a slice of 2D slices.

For each thread, the `distributor` function calls the goroutine `executeATurn`, handing it one of the slices returned by `splitWorld` and a channel for it to send its result to. This use of channels allows the worker functions to operate and return their results separately, so that they can then be combined.

To ensure that all of the workers have finished processing before the results are combined, a WaitGroup is used, which waits for each goroutine to complete before it moves on to combining the results by retrieving what has been sent to each channel.

`executeATurn` itself defers the WaitGroup until it is completed, performs the `calculateNextState` function on the world it has been given, and then sends the world to the `results` channel it has been passed.

`calculateNextState` is the same function used in our original GoL logic - given a world represented by a 2D slice of bytes, it will calculate the new state, following the given logic that:

- any live cell with fewer than two live neighbours dies
- any live cell with two or three live neighbours is unaffected
- any live cell with more than three live neighbours dies
- any dead cell with exactly three live neighbours becomes alive

It does this by using a separate function, `findAliveNeighbours`, to find the living neighbours of any given cell, so that comparisons can then be made and the new state of the cell can be calculated, if there is a change.

Attempted Development

As suggested by Step 3 of the coursework description, we implemented a ticker to report the number of alive cells every two seconds, using the `AliveCellsCount` event. We created a `backgroundTicker` function, to be run as a goroutine within `distributor`.

In order for the correct event to be sent to the channel, it required the use of two new channels: `tickTurn`, which would be used to retrieve the current turn, necessary as a parameter for `AliveCellsCount`, and `tickFinish`, which would be used to stop the ticker when all the turns had been completed.

The ticker was called once the original world slice had been created and the initially alive cells had been flipped, and stopped by a signal sent to the `tickFinish` channel after all the turns had been completed. At the beginning of each turn, the turn number was sent to the `tickTurn` channel. Both channels used were closed after the ticker was stopped, to confirm that all necessary data had been sent.

Despite implementing all this logic in a way that we believed would work, actually attempting to run the tests on this code led to timing out, even when using the `-timeout` flag and increasing the duration considerably. We believe this is due to a deadlock somewhere in the code, but were unable to find and fix this. As a result, in order to run the tests, we removed all of the code we had written for the ticker.

```
func backgroundTicker(c distributorChannels, world [][]byte) {
  ticker := time.NewTicker(2 * time.Second)

  for _ = range ticker.C {
     select {
     case <- c.tickFinish:
        return
     default:
        turn := <- c.tickTurn
        c.events <- AliveCellsCount{turn, len(calculateAliveCells(world))}
     }
  }
}
```

Figure 1: Removed backgroundTicker function

We encountered a similar problem when attempting to complete Step 4, outputting the state of the board after all the turns had been completed. This process worked in reverse of the for loops that initially retrieved the input and populated the world, looping through each cell in the world and then sending it to the `ioOutput` channel. When the code was run with this addition, the tests ran incredibly slowly, leading to timeout.

As this is the same as what happened with the `backgroundTicker`, it's likely that this problem is caused by a deadlock with the channel used.

```
for i := 0; i < p.ImageHeight; i++ {
  for j := 0; j < p.ImageWidth; j++ {
     c.ioOutput <- world[i][j]
  }
}
```

Figure 2: Removed code sending output to ioOutput channel

Distributed Implementation

We did not attempt the distributed implementation due to lack of time, as we decided that completing the parallel implementation would be more productive than moving on with it unfinished.

However, as we encountered more issues than expected with the parallel approach and did not complete it, in retrospect, it may have been worthwhile at least starting to develop the distributed implementation.

Critical Analysis

Analysing Test Results

In the final iteration of our code, 55/144 of the `TestGol` tests pass, and the `TestAlive` and `TestPgm` tests do not run, due to the errors we encountered in implementing these steps.

The tests that do pass include:

- Those that do not perform any turns - the code successfully retrieves the original image and adds this to the world.
- Those that evolve the Game of Life with one thread, regardless of dimensions or number of turns - this implies that the fault in the code relates to the splitting of the world for the threads, not to the Game of Life logic or the returning of results

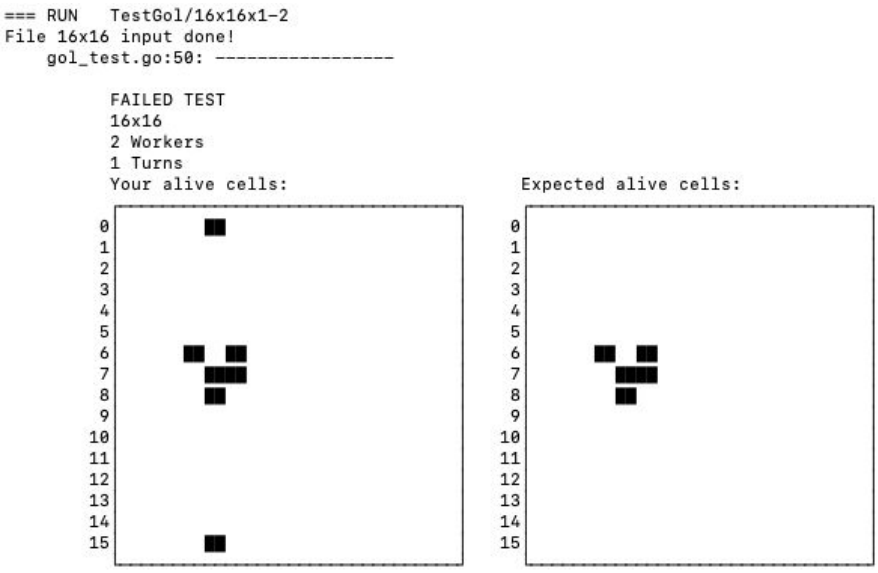

Figure 3: Result for TestGol 16x16x1-2

In Figure 3, it can be seen that although the “correct” pattern of alive cells is visible, there are additional erroneous alive cells shown. These may be caused by additional cells being marked as alive during the process of splitting the world into threads.

These problems propagate across the world over the evolution of turns, meaning that the tests run with the same number of threads, but a higher number of turns, produce results considerably further from the expected alive cells, as seen in Figure 4.

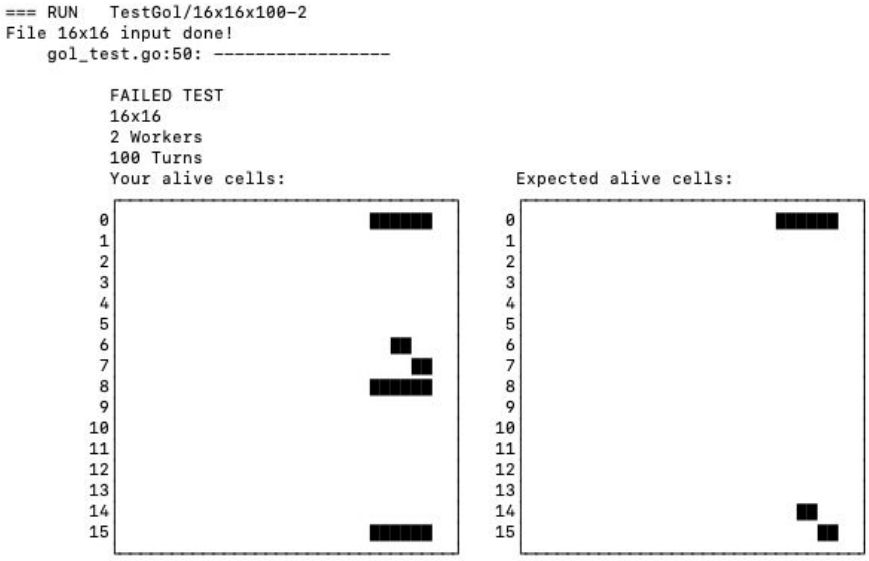

Figure 4: Result for TestGol 16x16x100-2