University of Bristol

Concurrent Computing

COMS20001

BsC Computer Science

Game of Life in XC

Author 1:
Josh Hamwee

Author 2:
Aidan Hood

Email:
jh17602@bristol.ac.uk

Email:
ah16165@bristol.ac.uk

December 6, 2018

Contents

1 Functionality & Design

1.1 Overview

In short, the functionality of our system revolves around the concept of splitting the game world into 4 equally sized quadrants. Each quadrant is processed in parallel by a unique worker thread, connected to the distributor function by a channel array with 4 channels - one for each worker. Each worker is sent the information for the quadrant it is processing, as well as an extra layer of cells surrounding the quadrant so that it has the relevant information needed to process the edge cells to it's quadrant. Once the worker has processed the quadrant, it sends back to the distributor only the $\frac{1}{4}$ world quadrant that it was assigned to process, and the world is reassembled in the distributor once it has received all 4 $\frac{1}{4}$ world quadrants back. This continues infinitely, but the next round of processing only starts once all 4 workers have returned their quadrant, and the distributor has reassembled the world state. Below is a diagram that illustrates this system, as well as the entire design as a whole.

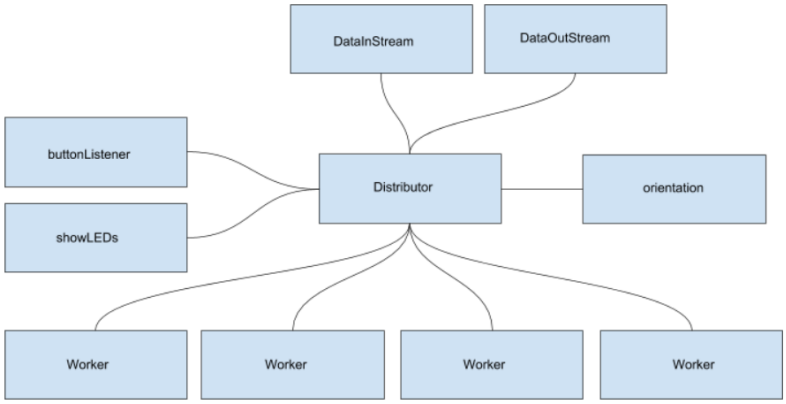

Figure 1: System Diagram

1.2 Non-Skeleton Functions/Features

- **waitMoment()** - A simple function that allows us to halt program flow for 0.7 seconds. This was useful in situations such as when a button press was being registered too quickly so was double registering.
- **overlap(int i)** - A function used so that if the world index was out of bounds of the world size then it overlapped back round to keep a closed system.
- **CellNewState(uchar workingWorld[3][3])** - A function that takes in a 3x3 matrix of cells, centred on the target cell. It scans through the cells and counts the number of alive cells adjacent to the target cell, and then carries out the game logic appropriately, returning the state of the target cell.

- **worker(chanend fromtoDist)** - The worker receives it's quadrant of the world,a s well as the extra layer of adjacent cells from distributor. Then, for every cell we create a 3x3 matrix surrounding the cell and call CellNewState on the matrix. The result is sent back to distributor.
- **Timers and board functions** - A timer system was implemented in distributor, and all LED/accelerometer features were implemented.

2 Tests & Experiments

Within our testing and experiments we break up the sections into testing the output images, testing the processing speed of our program and finally the limitations of our system after all our experiments have taken place.

2.1 Outputs

The most important factor in the testing is checking whether the output images of our system are correct. The first way of doing this is by running the 16x16 image, called test.png, for two rounds and then outputting it. If the image is correct to what we anticipated/calculated before then we know that the outputs are working.

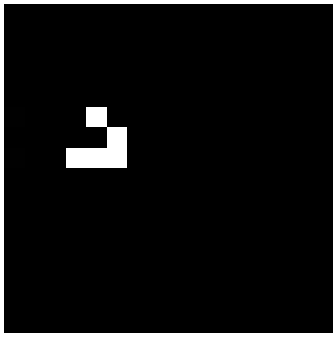

Figure 2: Original

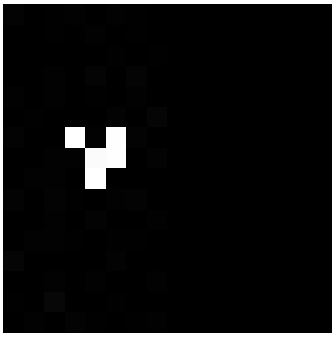

Figure 3: 1 Process

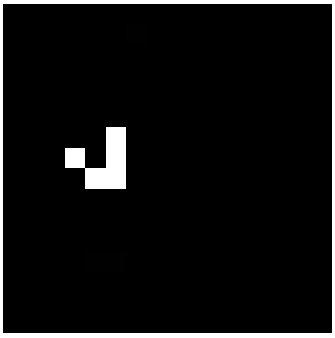

Figure 4: 2 Processes

Above in the first three figures, you can see three different images showing the processing of the standard 16x16 image given in the original download of the skeleton code. After working our what the states should be in figures 2 & 3 we can safely say the our game of life logic is correct. Figure 6 was our own test when we created a personal 16x16 world using GIMP. The image shown is the result over 3 processes.

2.2 Timings

The second test in which we undertook was checking the processing speed of our boards over 100 processes. We wanted to see how efficient our concurrent worker threads were

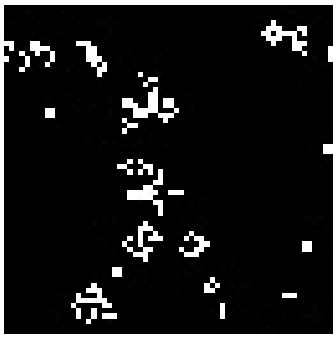

Figure 5: 64x64

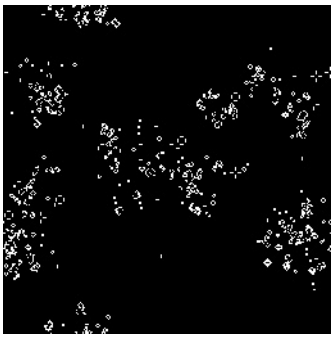

Figure 6: 256x256

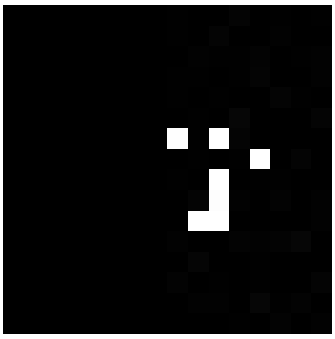

Figure 7: Personal

working in comparison to no workers. As you can see, from the table below. The addition of 4 workers has made the processing speed considerably faster.

Size	Workers	No Workers
16x16	1	13
64x64	1	17
128x128	5	39
256x256	20	N/A

Table 1: Time taken with workers vs no workers

There were a few anomalies with these results. Firstly we were surprised that the 16x16 and 64x64 sizes processed in the same amount of time. We believe that the 16x16 size would have processed faster, however due to how we used integers as the timers value we only see it as 1 second for both. We could not process the 256x256 size on the board with no workers, due to how we created the program. It was not able to compile.

2.3 Limitations

Our program does have a few limitations, and we know what is responsible for it. Firstly we cannot process any images larger than 256x256. This is due to us not considering bit packing until our last hurdle. We did attempt it for a long time, but due to the way we developed the rest of the project we were not able to implement it. Secondly we were not able to implement 8 workers into our project. This was because we split the board up into quadrants rather than lines. We will discuss these points later in our report.

3 Critical Analysis

3.1 Performance

Our tests show that our performance for small files is excellent, with 100 processes being completed in 1 second. We anticipate that the 16x16 files process even faster, however 1 second is the minimum our system was able to record. For medium sized files of 128x128 and 256x256 the results were still very good with 100 processes taking 5 and 20 seconds respectively. Although good, these could be quicker and listed below are some improvements that we anticipate would make the process faster:

- More workers - If we split the board into more workers working in parallel then it would inevitably speed up the processing time. We think the best way to implement this would be a change of system however, and move away from splitting the world into quadrants/sectors, and allow one worker to take on one line of pixels.
- Workers on dedicated cores - Assigning the workers specific cores could speed up the processing time if we gave each worker a dedicated core.
- Concurrent Workers vs Parallel Workers - Our system is concurrent, but the workers themselves run just in parallel. If we had designed the system so that they were communicating with each other and constantly updating their segment's state that would shave off time as the distributor would not have to reassemble and then resend out the world every processing round.
- Parallel processing in distributor - We send and receive the world's to each worker in loops. This could be improved by having each worker send and receive their world in parallel.

3.2 Image Size

The biggest limitation to our system was that we could not process images of size 512x512 or above. This was due to us designing and implementing the system before we knew that bit packing was essential in order to be memory efficient. We discovered this in the last week, and as we were both unfamiliar with bit-packing, and our system would have to had been completely redesigned we were unfortunately unable to implement it. We did however come out of it with a good grasp of the bit-packing process, despite our system not using it. In the ZIP file submitted there is a version of our Game of Life with our attempted bit-packing implementation. We were able to write macros for the functions needed to bit-pack, we just struggled with adapting our system that dealt with bytes to deal with bits.