# Game of Life Report COMS20001

**By Michael Mafeni (mm17291) and Ainesh Sevellaraja (as17284)**

## Functionality and Design

We implemented the game of life automation by splitting the grid into 8 parts vertically in our distributor thread and delegating each part to a worker thread. This was done via replication by declaring the workers' channels for communicating with the distributor as an array of channels. Each worker also communicated with the previous worker and the next worker to obtain extra columns. These are necessary to determine the neighbouring cells of the left and right most columns of that part of the grid, so that the rules of Game of Life can be applied to them. The rules are then applied to all cells continuously until the SW2 button is pressed. When this is pressed, each worker sends their part of the grid back to the distributor, where the grid is assembled and then written to the image.

We used message passing via the use of synchronous channels for communication between the threads. All the worker threads were delegated to run on tile 1 with everything else running on tile 0. Overall, we used 14 channels and 16 cores.

Two extra threads were created to model the button behaviour and the LED behaviour. The button thread sent data to the distributor notifying it of what button was pressed and the LED thread received a bit pattern from the distributor which was then sent to the appropriate port.

By implementing the 8 worker threads, the distributor does not need to wait for each the rules to be performed on each cell sequentially, as the worker threads run concurrently. The use of message passing between workers made it unnecessary to send the grid back to the distributor after each iteration, which sped up the processing time. Also as each worker got their dummy rows from the other worker, this made the code much easier to write as you would have needed to worry about modulus operations if you sent the extra rows via the distributor. Deadlock between communicating workers was avoided by first making the workers with even IDs send while the ones with odd IDs receive and vice versa.

The xc timer unfortunately could only time up to approximately 42 seconds. We solved this problem by creating a timer thread whereby if the value was greater than 42.92, we add it to a variable storing the no of times this occurred. We would then add them all back before sending it to the distributor. This timer was also used to help time our experiments.

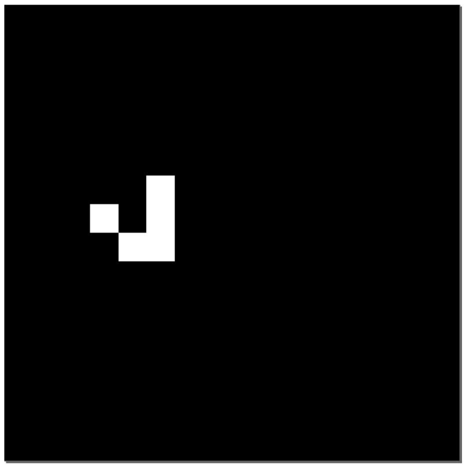
## Tests and Experiments

We tested many methods that we could have implemented the game of life algorithm. In these experiments we varied the image size and the number of worker threads. We then recorded how long it took for 100 iterations 5 times per image and got an average.

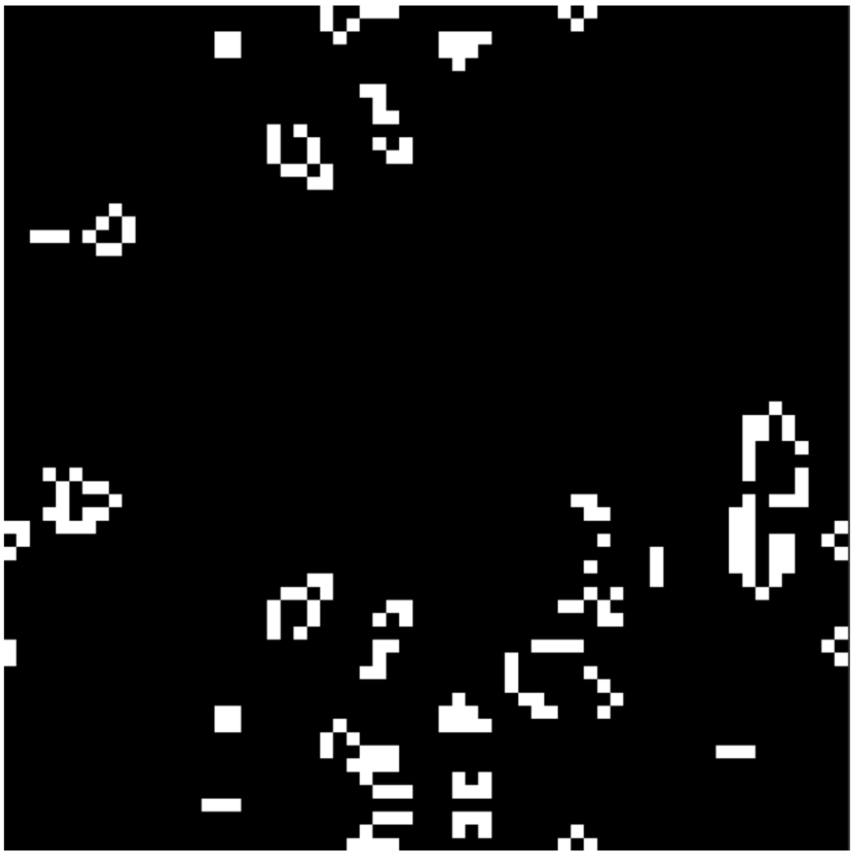
| | **Game of Life After 100 Iterations** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Image Size (by pixel)** | **Time/ms** | | | | | |
| | | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| 4 workers (distributor sends extra columns) (synchronous) | 16X16 | 209.42 | 208.05 | 209.85 | 211.66 | 209.69 | 209.73 |
| | 64x64 | 1531.46 | 1531.19 | 1529.04 | 1524.45 | 1532.36 | 1529.7 |
| | 128x128 | 5623.02 | 5623.99 | 5625.72 | 5623.99 | 5618.48 | 5623.04 |
| | 256x256 | 22088.45 | 22089.1 | 22089.1 | 22085.33 | 22087.7 | 22087.94 |
| | 512x512 | | | | | | |
| | | | | | | | |
| 8 workers (distributor sends extra columns) (synchronous) | 16X16 | 189.86 | 190.84 | 188.31 | 189.93 | 187.32 | 189.25 |
| | 64x64 | 1177.47 | 1178.87 | 1177.52 | 1177.73 | 1176.14 | 1177.55 |
| | 128x128 | 4315.85 | 4320.29 | 4323.81 | 4319.08 | 4320.17 | 4319.84 |
| | 256x256 | 16586.76 | 16587.16 | 16586.13 | 16587.1 | 16586.06 | 16586.64 |
| | 512x512 | | | | | | |
| | | | | | | | |
| message passing between 8 workers | 16X16 | 161.98 | 160.97 | 155.23 | 161.58 | 164.74 | 160.9 |
| | 64x64 | 854.76 | 851.46 | 854.47 | 854.12 | 855.88 | 854.14 |
| | 128x128 | 3077.23 | 3074.75 | 3076.95 | 3076.58 | 3076.95 | 3076.49 |
| | 256x256 | 11748.46 | 11744.96 | 11748.65 | 11754.67 | 11748.7 | 11749.09 |
| | 512x512 | | | | | | |

Factors affecting our Implementation

a) No of worker threads = As you can see above using more worker threads led to faster times. However there was a limit as to how much we could increase this by. For one, the XMOS board only contained 16 cores and with each worker thread being run on a separate core, this was hard to increase.
b) Storage space: Each worker stored their part of the grid plus two extra columns in a 2D array. Although this made the implementation easier to understand, this took up a lot of storage space and as a result we did not have enough memory to run a 512 x 512 grid



A 16 x 16 image of game of life after 2 iterations



A 64 x 64 image of game of life after 100 iterations

## **Critical Analysis**

Our system can work on images up to 256 x 256 bytes and evolves the game of life by:

a) 1.60ms/iteration for an image of 16 x 16 bytes.
b) 8.54ms/iteration for an image of 64 x 64 bytes
c) 30.76ms/iteration for an image of 128 x 128 bytes
d) 117.49ms/iteration for an image of 256 x 256 bytes

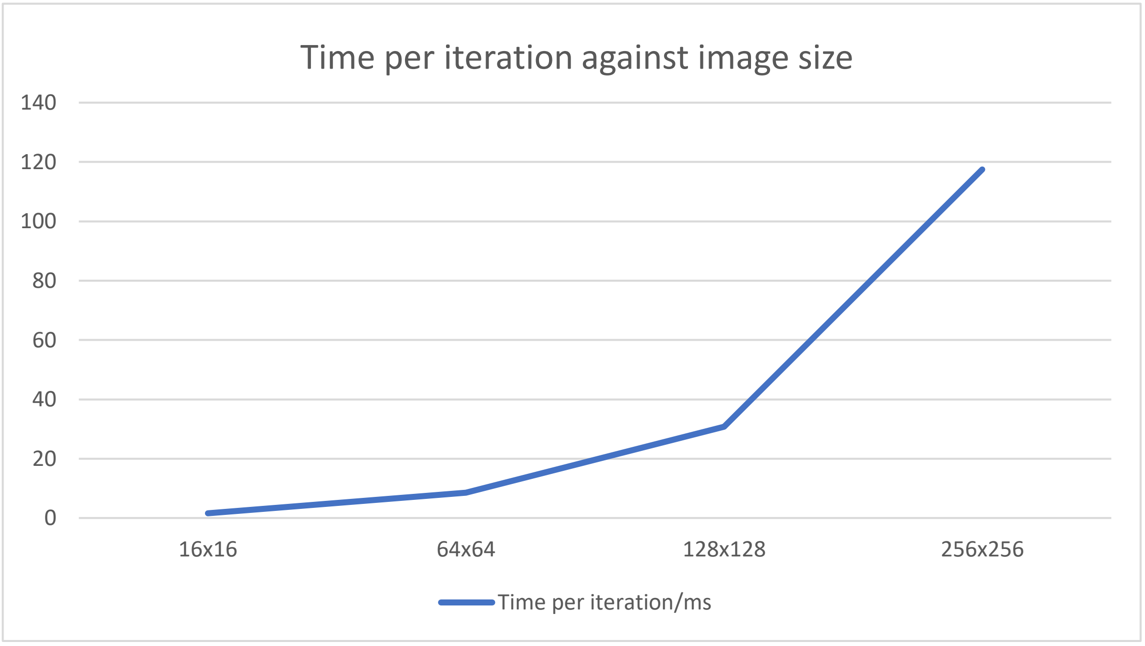
Overall, our system was relatively fast but it could be improved. One bottle neck in our code was the use of a 2d array to store the grid in worker. Although this seems intuitive, this made it impossible for us to work on images larger than 256x256 bytes because there wasn't enough memory on the board. As the only values being stored are 255 (alive) and 0 dead, we could instead make use of bit manipulation to store a part of the grid as a single numeric value.

Also, our code only works for images which have widths divisible by 8. To account for this we could have made the final worker take in the remaining columns. Also, although we made use of all the cores on 1 tile, there was still memory available to work with on tile 0 and we could've delegated more workers to that tile.

Furthermore, our timer could be improved. The way our timer currently works is that if the timed value is greater than 42.92, we add it to a variable storing the no of times this occurred and add all the values back. However, a more precise way we could have gone about it is record the time taken per iteration adding it to a total value each time.

The use of asynchronous channels could have made our system faster but we could not implement because we had too many streaming channels in a tile.

Overall our system worked well but could be improved. This project has helped us to learn the basics of concurrent computing via the message passing model.


Time per iteration against image size
16x16
64x64
128x128
256x256
Time per iteration/ms