Andy Thorn at17556@my.bristol.ac.uk (BSc)
Roberto Lau-Soto rl17185@my.bristol.ac.uk (MEng)

Concurrent Computing CW1: Game of Life

Functionality and Design

We made the Game of Life using features of the xCore-200 Explorer board such as the multi-core processor tiles, the orientations sensors, the LEDs and the buttons. We used 8 worker threads working concurrently to process the image input. The game is started using button SW1 to start the reading of the input image, then the board processes the next iterations of the game. The game is finished when either it reaches 100 iterations, the user tilts the board (which pauses the game until the board is returned to its original position and also prints the status report, thus displaying the time taken, number of rounds elapsed and number of living cells) or the user presses button SW2 which exports the current game state to “testout.pgm”. The LEDs also light up according with the colours given in the assignment brief, including flashing green for every round that is being processed.

The program first starts by declaring all of the channels that are to be used for synchronous message passing in the system; an example of such channel would be the channels used between the workers and the farmer. We then make use of a par statement to simultaneously execute and run all of our required threads concurrently. We further improved the efficiency of the program by running the threads on different tiles: all of our worker threads execute on tile 1, and all of the other threads (buttons, orientation, image reading and writing, distributor) execute on tile 0.

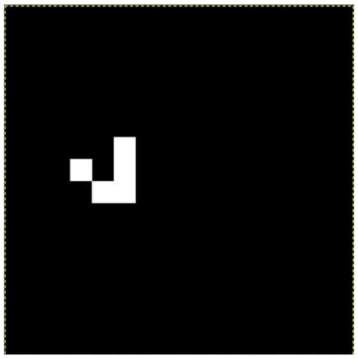
The overall program makes use of a farmer-worker model to implement the game, with the distributor acting as the farmer. The distributor is responsible for handling the timing, button inputs, LED outputs and tilting/pausing the game; as well as the breaking down of the image into rows and feeding them to the workers. The algorithm we use to break down the image is very complex and took a long time to do. It was essential that this algorithm was developed and tested carefully as it acts as the main connection point between the farmer and the worker. It’s purpose is to break down the board into rows and allocate the rows to the worker threads. It was very important that the image was split correctly every time for every image size as the right data had to be sent to the workers in order for them to work synchronously together. The number of lines allocated to each worker is calculated by taking the image height and dividing it by the total number of workers. Each worker is then fed the specific lines of the image it must implement the game logic on via a channel. An additional two ‘ghost’ rows are also sent to the workers as they are required in the process of checking the number of live cells in the edge rows.

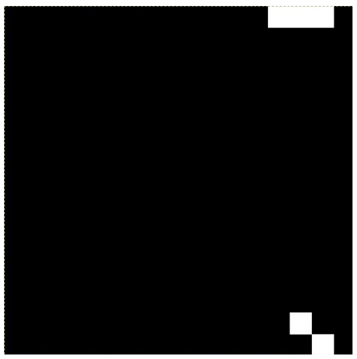
Once the worker receives the part of the image it has been allocated, it then executes the game logic on the image and changes the cells to living and dead accordingly. It then feeds back the new edited picture back to the distributor. All of the worker threads run concurrently so the analysis of the image is a lot faster then just using one worker. Once the distributor collects all the sections of the image from the workers, it uses an algorithm to patch it back together and stores the new image inside an array which is then ready to be written into a new pgm file.

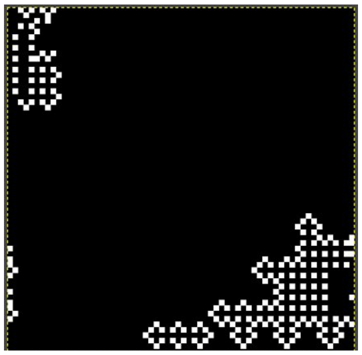
Tests and Experiments

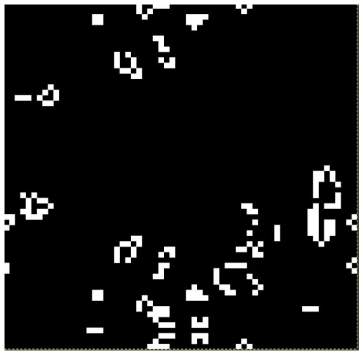
The images show the output of the game after 2 iterations and 100 iterations (left to right respectively). The table below shows the result of the timer after the given test images have been processed and iterated over.

Image size	Time taken after 2 iterations(ms)	Time taken after 100 iterations(ms)
16x16	317	10522
64x64	385	14243
128x128	536	22935
256x256	1187	54460

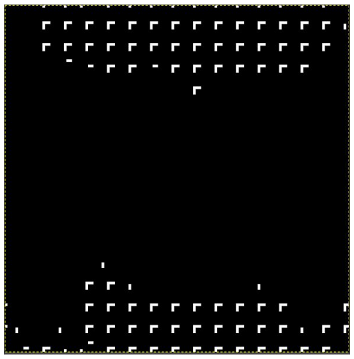

16x16 (2 iterations)

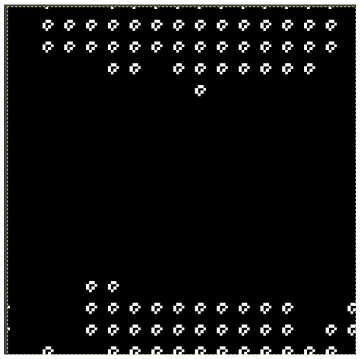

16x16 (100 iterations)

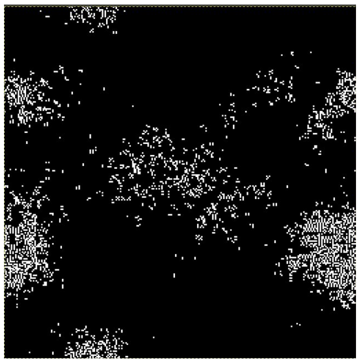

64x64 (2 iterations)

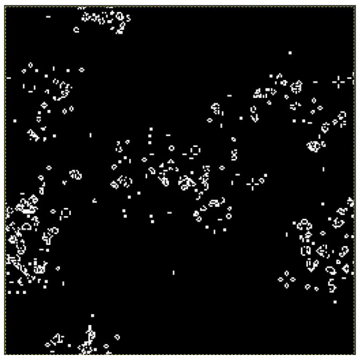

64x64 (100 iterations)

Tests and Experiments


128x128 (2 iterations)


128x128 (100 iterations)


256x256 (2 iterations)


256x256 (100 iterations)

Since we were unable to implement bit packing our program would run out of memory when we tried to do 512x512 sized images and above. The amount of memory we used was 542,120 bytes, however the memory available was 262,144 bytes. Bit packing would have solved this by increasing memory efficiency by 8 times, due to storage of a pixel as a single bit instead of an 8-bit sized byte. An observation we made was that changing the number of workers between 8 and 4 had very minimal effect on the time taken, potentially explained by the size of the images not being large enough to notice a significant difference.

Critical Analysis

With the benefit of hindsight we would have liked to change some things about our implementation. Considering that the largest image input accepted is 256x256, we definitely would've wanted to improve our farmer-worker model to be more efficient to handle larger images like 512x512 and beyond. We attempted to use a server-client interface for better communication to handle the workers, however we couldn't get it to work and settled on using channels, which could affect the efficiency of our system. However since channels allow for two-way communication we consider it to be a suitable alternative.

Another improvement would have been to use bitpacking as advised to effectively to deal with the limitations of the memory of the board, as currently each cell of the image is stored in a byte instead of a bit (where we could have referenced the state of the cell using 0 or 1) which is wasteful and causes the storing and processing of the image to take too long considering the alternatives. Reflecting on our work strategy, we should have considered working on prioritising this earlier, however we underestimated the complexity of the task. As a result we learnt that the reading and outputting of larger images by our system is outperformed by systems built by some of our peers. With more time we also would've liked to compare using synchronous versus asynchronous channels for communication between the distributor and workers, for example testing the use of multiple streaming channels (using a buffer to control the data exchanges) and trying out transaction sequences (using 'master and slave' threads which can run concurrently). We would have also liked to have experimented more with making our own, larger images and observing the results given different initial conditions.