COMS20001: Concurrent Computing

Conway's Game of Life

Louis Heath - lh16421@my.bristol.ac.uk
Jack Jones - jj16791@my.bristol.ac.uk
MEng Computer Science

Functionality and Design

We have implemented a simulation of Conway's Game of Life, developing the game state continuously with eight concurrent worker threads. The program utilises hardware features of the xCORE-200 eXplorer, requiring button input 'SW1' to begin, input 'SW2' to output the current game state and also providing recognition of the tilting of the orientation sensor which pauses the game, including the timing, and prints information about the current game state. This information includes the time elapsed, the iteration number and the total number of alive cells. The onboard LEDs are also used to identify when the board is reading, processing, paused and writing.

Though adding complexity to the task, we decided it was appropriate to pause the timing both whilst the board is tilted and also whilst the board is outputting the game state to a '.pgm' file. Without this functionality the time recorded is not directly connected to the processing time.

Our implementation uses a farmer-worker pattern, with the farmer being our '*distributor*' function. The farmer manages input, output, work distribution, pausing and hardware functionality. The workers receive their portion of the game state from the farmer and evaluate it indefinitely, each turn informing their neighbouring workers of newly calculated 'edge' rows required for the next iteration. These two 'edge' rows are the rows above and below a worker's portion of the game state. They are vital in determining the next state of any cell on the edge of a worker's portion, as this depends on how many living neighbours it has.

To do this we altered the classic farmer-worker star structure to include communication between workers, making our system more coarsely grained. The alternative would be for the farmer to receive the entire game state from the workers each and every iteration, which would be significantly slower. The function which manages worker-to-worker communication is designed carefully to avoid deadlock.

The *distributor* is central to the pausing logic in that it receives input from the buttons and orientation sensor in order to dictate which of three states the workers should take. These include the default state, in which workers process the next iteration, and two pause states. The first pause state triggers when the board is tilted, in this case each worker updates the *distributor* with current game statistics. The second is when an output file has been requested, requiring the workers to send their portion of the game state back to the *distributor*.

We have utilised bit packing to combat memory limitations when processing larger images. Each cell state originally occupied an entire 8-bit *uchar* which is wasteful, given there are only two possible states. We use bitwise operators to fit eight cell states into one *uchar*, reducing our array sizes eightfold.

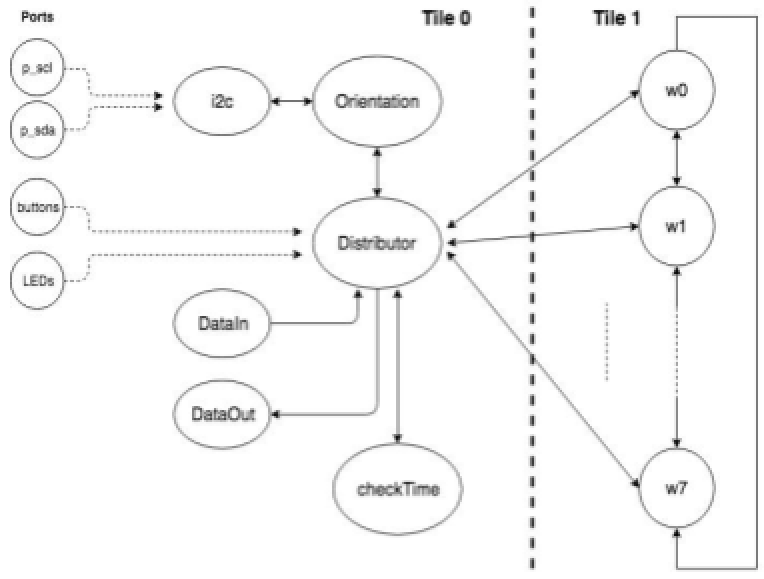

Figure 1: Relationship diagram

Tests and Experiments

The pairs of images on the right are the game states after 2 and 100 iterations of our implementation of Conway's Game of Life on the provided test files. The time taken for each output when processed by eight workers can be seen in the table below.

	Time after *i* iterations (ms)	
Image Size	***i* = 2**	***i* = 100**
16x16	2	64
64x64	20	1032
128x128	84	4150
256x256	324	16029
512x512	1289	64462

Figure 7: Table of processing speeds for varying game images

After repeating these tests with four workers and also our sequential implementation we were able to construct a graph (Fig. 8) displaying the relationship between the number of pixels in a game image and the time taken to process 100 iterations. The full data set and subsequent calculations can be found in our 'timings' spreadsheet, attached with the project.

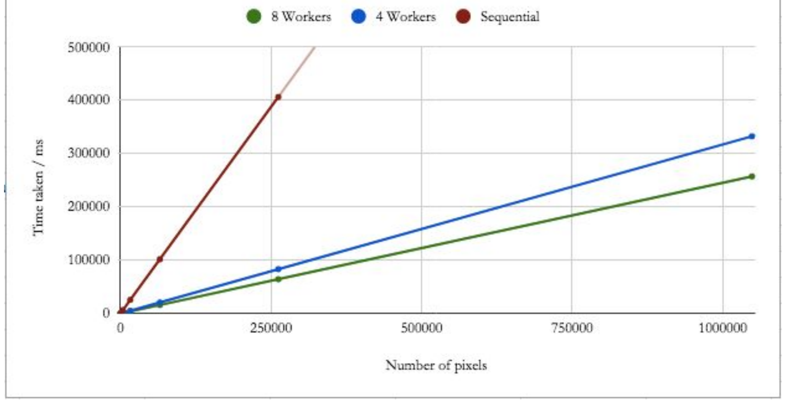

Figure 8: A graph to compare processing speeds of concurrent and sequential implementations over 100 iterations

It is evident from the gradient of each line that the more parallelised systems processed more pixels per second. From the data collected we could calculate the difference in performance between each system, in particular their processing speed. On average, to process one iteration on one pixel, the eight worker system takes 0.0025ms, the four worker system takes 0.0032ms and the sequential takes 0.016ms. These findings tell us that the eight worker system is around 6.3x faster than the sequential and 1.3x faster than four workers.

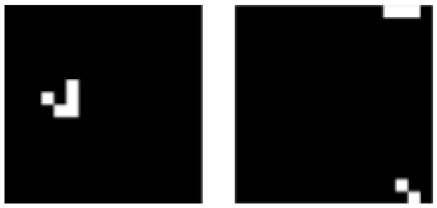

Figure 2: 16x16

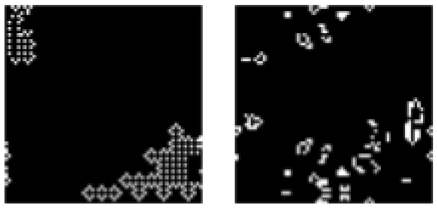

Figure 3: 64x64

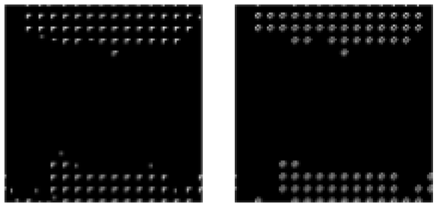

Figure 4: 128x128

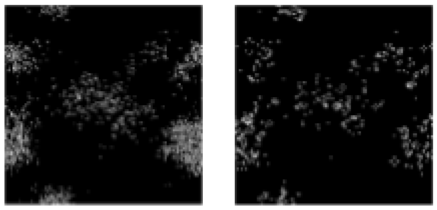

Figure 5: 256x 256

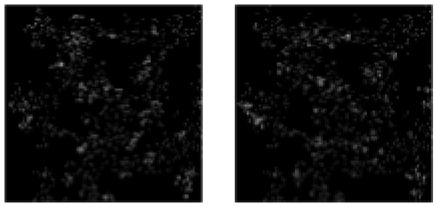

Figure 6: 512x512

Before gathering the times in Figure 7 we tested a series of optimisation attempts. After each modification we checked for a change in the time taken to iterate 100 times on a 64x64 image, initially taking 1989ms. We began by looking at the helper functions in the innermost loop of our *worker* function. The first improvement was made in our modular arithmetic unit '*mod*', which we had hastily developed during week 3. By replacing wasteful *if* and *else* branches with a basic *while*-loop we reduced the test time to 1384ms. We then decreased it further to 1033ms by reducing memory usage in our '*getNeighbours*' function, replacing an array with a nested *for*-loop. Not all optimisations were successful, for example we tried to improve our bit-packing function by calling *return* early, which we discovered to be slightly slower. We also experimented with the cost of sending different data types over channels, for example sending an 8-bit *uchar* rather than a 16-bit *int*. This ended up somewhat slower, possibly due to the type-casting required before transfer.

Though our timing tests above provided a fair measure of speed, we also wanted to test how efficiently we manage memory. To gauge our performance we needed to find our system's upper limit, requiring us to be able to create game images of custom size. Our first solution was to implement *pgm_generator*, in which an 8x8 pixel pattern was repeated to populate a larger image for output. This program was time consuming and provided limited flexibility in image design, only allowing repeated tiles. Consequently we decided to draw images ourselves using a graphics editor such as "GIMP", then converting them to '.pgm' files with an image viewer such as "IrfanView". Though far more time efficient, every cell in a produced image was dying within the first two iterations, due to the thick brush size in the editor. To solve this we reused the infinite cell layout seen in Figure 2 as a building block, allowing us to construct images that will stay alive. We found our upper limit to be 1184x1184 when using eight workers, and 1208x1208 when using four.

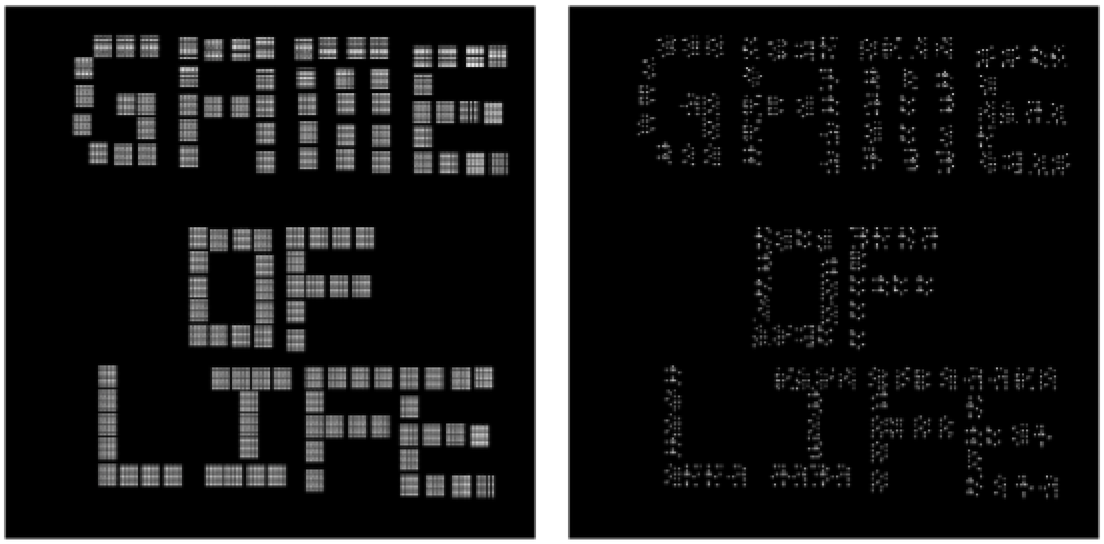

Figure 9: 1184x1184 after 2 iterations

Whilst developing our system we added automated testing for all of our non-void functions, in particular our modular arithmetic, the packing and unpacking of bytes and the counting of living cells neighbouring a given cell. These tests provided peace of mind when editing connected logic as we knew they would not only catch our mistakes but tell us where they are.

Critical Analysis

To benchmark the performance of our system we note that the maximum input size accepted is 1184x1184 with eight workers, and 1208x1208 with four, as discussed above. With regards to speed, our system processes 100 iterations of 512x512 in 64.5 seconds, and on average completes one iteration on one pixel, in 0.0025ms. This scales linearly, as shown by the constant gradient in Figure 8.

After comparing processing speeds with other teams we have learnt that our system is quite often outperformed, however when comparing maximum input files we think our system fares relatively well. This leads us to conclude that our system has room for optimisation, but stores the game state efficiently. In this section we explore potential modifications that could benefit our implementation, as well as our decisions to not utilise certain features.

A functionality provided by xC which could have improved our system but which we avoided using was pointer declaration. By using pointers we could have potentially decreased duplication of memory at the cost of added complexity, as avoiding race-conditions and controlling critical-sections. Though on a few occasions we attempted declaring movable pointers to sections in the game array, xTimeComposer always responded with new error messages.

To improve communication we considered using server-client interfaces which are often more readable than channels. They can also avoid confusion, for example when more than one entity is affecting a binary state. In the case of our game, both the tilting of the board and the pressing of 'SW2' toggle a binary pausing state in *distributor*, but a local pause state also exists in *orientation*. If communication is handled with channels this can become complex, for example if the board is tilted, thus paused, and during this state 'SW2' is pressed, *distributor* needs to not unpause the board and also inform *orientation* not to change its local pause state. An interface with *guarded* functions would simplify this as only the relevant functions are caught. Despite the advantage in this situation we decided against using interfaces, as they only allow transfer in one direction. The majority of our communication links are bidirectional, as seen in Figure 1, so we decided the readability advantages of utilising existing channels outweighed those of toggling pause states with interfaces.

Streaming channels provide the ability to send data between threads without waiting for it to be received. It does so by using a buffer. The main way we could have utilised this concept would have been for communication from farmer to worker. In our current system the farmer must wait for one worker to receive a value down a channel before it can communicate with the next, staggering the worker processing and delaying the farmer. Streaming channels would avoid waiting on both ends of the communication, freeing the farmer to continue checking for button input and preventing the delay when a worker receives its instruction signal.