Thomas Raybould, Computer Science MEng tr17544@bristol.ac.uk
Finn Wilkinson, Computer Science MEng fw17231@bristol.ac.uk

Functionality & Design

First, we implemented a bit packing function. As image data is read in, groups of 8 cells are packed into a single uchar by accumulating the cells values. If the position in the uchar is i = 1 -> 6, we only need to look at 3 uchar values to calculate the number of alive neighbours. If the position in row is 0 or 7, we look at 6 uchar values. Without bit packing, we would have needed to access 9 uchar values for each cell.

We used a single distributor thread and 8 concurrent worker threads to process the image. Each worker is sent data which represents an IMWD x (n+2) [1] grid of cells. The worker threads then return data which represents an IMWD x n grid of cells. The two extra rows are for calculating the neighbours correctly for the top and bottom rows of the IMWD x n grid. The distributor listens for data to be returned from the worker functions, and will accept data as soon as it is ready due to the use of select statements. We used asynchronous and synchronous channel communication to implement the 8 worker threads over both tiles. We found through testing different configurations that having 4 workers on tile[0] and 4 on tile[1] was the quickest distribution of workers, with the distributor being on tile[1]. This uses asynchronous channels for the workers on tile[1] and synchronous channels for the workers on tile[0].

To be able to process images up to 1024x1024 we had to improve our use of memory. In the distributor, we originally stored one full copy of the grid and two copies of the ‘bit packed’ grid. We were able to remove one of the arrays with ‘bit packing’ by overwriting the initial image values of the array with the processed values, rather than storing the processed values in a new array. Eliminating the original full image array was achieved by sending the ‘packed’ rows to DataOutStream, unpacking this row, then writing the row to the output file. This is repeated until the whole image is written to the output file. Removing these copies of the arrays meant there was enough memory on the stack to store larger images when they had been ‘packed’.

Both the button and LED functionality were implemented by using listener functions. For the buttons, a while loop was used to continually listen out for button presses. If either SW1 or SW2 was pressed, they would send a message down a channel to the appropriate part of the program, implementing the desired functionality that was specified. Setting up the LED function was similar. In a while loop we used a select statement to listen, in theory, to multiple channels consecutively. Each channel was linked to another function that sends the correct value dependant on which LED was to light up. This int was then sent to the LED port on the board which lights up the correct coloured LED at the specified point in processing. We extended the orientation code provided to ensure that the program will pause if the board is tilted along its x axis in either direction, and that when the board is un-tilted, the program will resume from where it was paused.

To keep track of time elapsed so far, so that this can be displayed when the program is paused, we created a timer function, run in parallel to everything else. When the program starts processing the first round, the distributor will send a piece of data to the timer via a channel to initialise the time against one of the boards built in timers. From here, a while loop and select statement is used to measure the time elapsed in 1/1000ths of seconds. Every period of 100000, we add 1 to a counter inside the timer function. When the program is paused, the timer is effectively paused, and the current time elapsed is sent to the distributor to display. Then when unpaused, the time variable is re-initialised and the counting continues. Implementing the timer this way decreases the occurrence and complexity of dealing with overflow errors.

[1] - The variable n is equal to IMHT / (number of workers).

Tests & Experiments

Test	Description	Time Taken
Test 1: processing 2 rounds imperatively	In this test we timed how long our first full implementation of the game takes when processing the 16x16 bit test.pgm over 2 rounds	1) 2.85 seconds
Test 2: Entire simulation of 2 rounds with 8-bit packing	We have extended our implementation to include some bit packing. 8 consecutive cells get turned into an unsigned char. On a 16x16 image.	1) 2.767 seconds 2) 2.843 seconds 3) 2.842 seconds
Test 3: Entire simulation (excluding I/O) of 100 rounds with 8-bit packing	No changes to the implementation. Processing 100 rounds now. 16x16 image.	1) 25.639 seconds 2) 25.360 seconds 3) 29.638 seconds
Test 4: Implemented two worker threads.	Now we have a distributor thread and 2 worker threads that do all of the processing. Used a 128x128 image.	1) 14.11 seconds 2) 13.886 seconds 3) 13.694 seconds
Test 5: Implemented 4 worker threads.	Increased the number of worker threads. Processing a 256x256 board.	1) 28.482 2) 28.389 3) 28.489
Test 6: Implemented 8 worker threads.	Increased the number of worker threads, processing a 256x256 board.	1) 23.565 2) 23.681 3) 23.568
Test 7: 8 workers on same tile, changed channels to array. No bit packing/unpacking in time as integrated into data in and data out, so that we could process 1024x1024.	The entire simulation of 100 rounds on a 256x256 board. Channels declared in an array, rather than individually declared. Bit packing moved to where data is read in. Workers the same.	1)17.068 2)17.311 3)17.061
Test 8: Removed all unnecessary print statements.	The configuration is the same as previous test, however a print statement that we used for testing the code has now been removed. Always	1)11.970 2)11.970 3)11.970

	printed 100 times, so no real effect on processing time.	
Test 9: 7 workers on one tile, 1 worker on other tile.	Entire simulation of 100 rounds on a 256x256 board. Now we have 7 workers on tile with distributor, to test efficiency of less cross channel communication	1)14.675 2)14.675 3)14.675
Test 10: Single worker on a tile with asynchronous channel	We have changed the distribution of worker threads across the tiles, and assigned one an asynchronous channel. Still a 256x256 grid	1)14.651 2)14.651 3)14.651
Test 11: 4 worker channels asynchronous, 4 synchronous.	4 worker threads on same tile as distributor have asynchronous channels, 4 on other tile don't. Still a 256x256 grid.	1)11.813 2)11.813 3)11.813
Test 12: 7 workers and distributor on one tile, other worker on seperate tile.	Tried having 7 workers on the same tile as the distributor function. Processing a 256x256 grid	1)14.805 2)14.805 3)14.805
Test 13: all workers with asynchronous channels (3 on tile 0, 5 on tile 1)	Only way we could implement all worker channels asynchronously, 256x256 grid.	1)11.867 2)11.867 3)11.867

Virtues:

- Splitting image into 8 equal parts, letting 8 parallel worker threads compute the new image
- Asynchronous channels when workers are on same tile to distributor
- Data packer enables messages to be passed quicker, and larger images processed
- Processed data received by distributor as soon as it has been calculated by a worker

Limitations:

- The number of asynchronous channels we can use. We would like to be able to implement an asynchronous channel between the distributor and every worker.
- There is a limit on the number of synchronous channels we can declare. If we were able to implement more, we could have implemented a pipeline, to increase the efficiency of the worker processing.
- Cannot have the 4 asynchronous workers on different tile to the distributor as it complains that too many streaming channels are declared. Doing it this way could improve processing speed

Critical Analysis

The maximum image size we have been able to fully process is a 1024 x 1024 bit image. As the size of the image is multiplied by 4, the processing time for the same number of rounds is also multiplied by roughly 4. It is not better or worse at processing larger or small images, and so is roughly linear.

Board	16x16	64x64	128x128	256x256	512x512	1024x1024
Time for 100 rounds	0.051 seconds	0.846 seconds	2.542 seconds	11.813 seconds	42.879 seconds	161.706 seconds

The first major improvements in processing time came from implementing multiple worker threads as seen from Test4 when it took less time to compute 100 rounds for a 128x128 image than it did previously for a 16x16 image. In the end we decided to use 8 workers, with 4 on tile[0] and the other 4 on tile[1], with the distributor also on tile[1]. We tested using asynchronous message passing in many configurations, but found that the quickest overall processing came when we used a mix, as seen in Test11; 4 worker threads using asynchronous message passing with the distributor, whilst the other 4 worker threads on the other tile use synchronous message passing. We could not use asynchronous message passing with the workers on tile[0] as we had errors with declaring too many streaming channels. This would have been ideal as the workers on the same tile as the distributor are secretly using memory sharing, which is more efficient than message passing. Where as the workers on the other tile must utilise message passing to communicate with the other tile. Using asynchronous channels for cross tile communication would allow for messages to be sent to the channels 'buffer' by workers as soon as they were ready, meaning they can get back to processing the data quicker.
We also had improvements from our bit packing, by streamlining the functions and integrating the packing/unpacking into the relevant I/O sections of the program. These are the most important factors in increasing our processing speed.

We think it would be possible to speed up the processing speeds by quite a bit. One way of doing this would be to detect large areas of the grid that have no alive cells. We could then have the distributor and worker functions completely ignore these areas during processing, as currently the program looks at every single cell, unnecessarily calculating the number of neighbours for large blank areas of the image. In the functions which calculate the number of alive neighbours, we could reduce how many times the array is accessed. For every group of 8 cells, you always need to look at the uchar they are stored in. It would be faster to create a copy of the word, then use the copy during calculations, rather than finding the array index you need every time. Another way we could improve the implementation is in the distributor function where it distributes the image to the workers. This is a long section with many if statements and modulus calculations to decide which worker to send a particular row to. Improving this would increase readability and could decrease distribution time.
If there were more threads and channels available to use, we could have implemented pipelining in the distributor and worker threads, meaning that as soon as data is ready to be processed, received, or sent it can be rather than waiting for all data to be available. However, to implement a 2-section pipeline in our system it would have required us to go back to 4 workers, as we would need 2 threads for each worker. Since we were already using the maximum number of threads in the system, this wasn't feasible.