Team 27 Course: Computer Science
Palvi Shah (MEng) - ps16925@my.bristol.ac.uk
Sibela Chinareva (BSc) - sc16913@my.bristol.ac.uk

Cellular Automaton Farm

Introduction

Over the past 6 weeks, we have implemented a concurrent version of Conway's "Game of Life" which uses memory on both tiles of the xCore200 and a number of worker threads. Our program explores parallelisation via farming incorporated with aspects of geometric parallelism. The xCore200 provides us with direct control of a parallel hardware platform and the language xC allows us to use multiple threads and point-to-point channel communication.

Functionality and Design

Buttons, LEDs, Orientation & Timing
LEDs and orientation were combined in a single thread using an array of two channels, while button input was handled as a separate parallel thread. Initially, the processing of the image is triggered when button SW1 is pressed and the board has been tilted. After that every tilt of the board results in the game being paused, which prints out a report of the current state and continues again when the board is horizontal. The LEDs are green for reading the image in, red for pausing, blue for exporting into a pgm file, which is caused by pressing the SW2 button, and flashing green for processing. *Select case* statements were used to ensure the processes run smoothly in a deadlock free way.

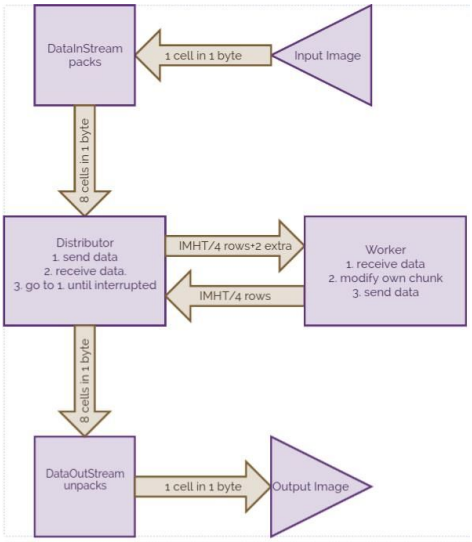
Timing
Timing was implemented using a special thread *whatTime,* located on *tile 0,* that has one single reference clock ticking at 100MHz. Once 100,000 cycles have passed, the time is incremented by 0.001 seconds. A separate thread for the timer allows us to keep track of the accurate time, independent of the other operations, as well as giving other threads access to the time at any point in the game. The time is printed out, to assist testing, every time the processing is paused or stopped.

Tile allocation
Our *main* has 6 threads working in parallel; one for each vital function. After experimenting with combinations of threads on our 2 tiles, we found that the best solution to achieve efficient synchronisation was to have the server and client threads for the orientation data on *tile 0* along with the *buttonListene*r thread. This allowed our system to be more responsive to the orientation data from the board. As we want the 4 workers to be parallel processes with a similar structure, we decided to have the worker threads on alternating tiles and implemented this using replication and nesting of *PAR*. This ensures that the concurrency of the workers is kept elegant. We declared an array of 4 channels to be used as synchronous, point-to-point connections between the distributor and each worker as well as 4 channels between the workers themselves.

Divided Board
After the PGM image is converted into a stream of pixel values, it is sent to the *distributor* thread and then to the workers who alter each pixel according to the rules of the Game of Life. Our **explicit process communication** is done via the ***Message Passing Model*** where the design involves dividing the image into 4 sections horizontally; each of which is allocated to one worker. Every worker, in addition to its section of the image, will initially receive the top row of its preceding worker and the bottom row of its succeeding worker in order to modify the pixels on the edge of its allocated section.

Palvi Shah (MEng) - ps16925@my.bristol.ac.uk
Sibela Chinareva (BSc) - sc16913@my.bristol.ac.uk

After the first iteration, the workers can modify their pixels solely by communicating between each other.



Bit Packing

In order to make more efficient use of memory and process large images, we implemented bit packing in *DataInStream*. We achieved this by packing every 8 pixels in a row into a single byte so that 1 bit corresponded to 1 pixel which in turn reduced the memory used by a factor of 8. The worker function originally performed bit manipulation on the unpacked bytes; however, to optimise our system further, we changed the worker logic so that it could manipulate the packed bytes, saving both time and memory significantly. This meant that the image would only have to be unpacked once - when exporting.

Tests and Experiments

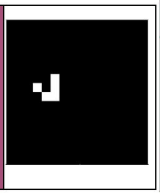
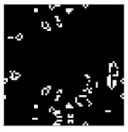
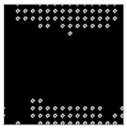
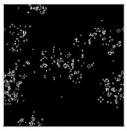
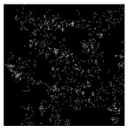
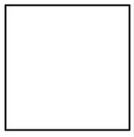
Test 1 - processing images with different number of workers

Our first test was to compare the processing speeds for different numbers of workers for 800 iterations. The hardest part of achieving the fastest possible computation was finding the right balance between number of worker threads and image size due to the possible communication bottlenecks. After experimenting with different values, we discovered that less workers result in faster processing of smaller images due to the need for less channel communication. However, as the image size increases, a worker needs to work on a bigger area, meaning it does more calculations. This then forces us to use more workers even though more time will be lost on message passing between channels. In the end, we noticed that the optimal number of workers was 4. After considering the memory usage of each number of workers, we concluded that dividing the workers evenly over the two tiles, alternating which tile they went on, was the most efficient configuration. The only case where this was not the optimal solution was on images larger than 512 x 512 because then the memory on *tile 1* becomes insufficient. We resolved this issue by putting 3 of the worker threads on *tile 0,* which then made it possible to process images of size 1024x1024.

	16 x 16	64 x 64	128 x 128	256 x 256	512 x 512
8 Workers	18.560	28.240	83.600	270.56	1062.08
4 Workers	18.560	28.320	74.280	277.280	1053.040
2 Workers	18.560	28.400	84.480	280.560	1065.920

Test 2 - processing images of different sizes

Our second test explored the difference in speeds of processing images of varied sizes. All of them were manipulated on by 8 workers for 100 iterations. Expectedly, the time processed increased with image size.

	Image Size	16 x 16	64 x 64	128 x 128	256 x 256	512 x 512	1024 x 1024
	Time taken	2.320	3.530	10.250	33.82	132.76	422.000
Image of 16 x 16 image after 2 iterations 	Image after 100 iterations						

Palvi Shah (MEng) - ps16925@my.bristol.ac.uk
Sibela Chinareva (BSc) - sc16913@my.bristol.ac.uk

Test 3 - comparing communication vs computation time

For this test, we timed how long one iteration takes for 2, 4 and 8 workers on 3 different board sizes to measure the speed of computation. We then measured the speed of communication between workers and the distributor by timing how long it took to send and receive data between them. From the results, it is evident that *as the number of workers increased, both the communication and computation time decreased.*
The communication time was significantly faster by around 5.5 times, for 8 workers on small images of sizes 64 or 16; however, 4 workers proved to have faster communications on larger images.

16 x 16	2 Workers	4 Workers	8 Workers
Communication time (s)	0.002836	0.002587	0.000513
Computation time (s)	0.0232	0.0232	0.0232
64 x 64	2 Workers	4 Workers	8 Workers
Communication time (s)	0.036954	0.002241	0.011484
Computation time (s)	0.0355	0.0354	0.0353
128 x 128	2 Workers	4 Workers	8 Workers
Communication time (s)	0.546048	0.533299	0.54132
Computation time (s)	1.0560	0.0934	0.1025

Test 4 - comparing the time for different ratios of alive:dead pixels

Our next experiment aimed to determine whether the initial map configuration influenced the computational time. We timed 4 workers for 100 rounds working on boards that were all alive, all dead, ½ alive, ¾ dead and an image where every adjacent byte alternated. The results in the table show that processing time depends on the proportion alive-dead pixels, as well as how evenly they are spread. For example, the difference in processing the all dead board and the board with alternating cells was nearly one second, 0.93299 in particular. Overall, we concluded that the closer the numbers of alive and dead cells are, the longer it takes to compute.

Image ratio	All alive	All dead	1/2 dead	3/4 dead	Alternating bytes
Time taken (s)	3.38159	3.08624	3.54731	3.55829	4.01923

Test 5 - improving on image size

Finally, we tried finding the largest image that our system can process. Despite the fact that bit packing did help us save memory and process the 512x512 image, we were hoping on improving our system even further. When trying to process images larger than this, there was still the issue of too much memory being used on one of the tiles. We solved this problem by moving three of the worker

Palvi Shah (MEng) - ps16925@my.bristol.ac.uk
Sibela Chinareva (BSc) - sc16913@my.bristol.ac.uk

threads on the other tile so that even amounts of memory were used by both tiles. Currently, we can process a 1024x1024 image with 8 workers for 4.22 seconds per iteration. Doing the worker calculations more efficiently, as well as experimenting more with organization of processes between tiles, may help us develop our program even more in the future.

Critical Analysis

Our program is efficient enough to process large images of up to 1024 x 1024 at a good speed of 4.22 seconds per iteration. The system can work on these large images due to us compressing the file from 8 bytes into 1 byte for processing and spreading the workers evenly over the 2 tiles, saving significant amounts of memory as well as time.

The virtues of our system came with some limiting factors such as the speed of computation. This is affected by the large amount of time taken to pass the board onto the workers, together with communication between each of the them. In the future, a possible improvement would be to store the board even more competently by packing 32 cells into 1 integer. This will improve the speed of communication between channels as well as make more efficient use of memory which will help us process images greater than 1024 x 1024 for shorter periods of time.

Another way to optimise the time taken for processing is to improve the worker logic. In the early stages of development, we were planning on unpacking the image in the worker, performing the game computations on this and sending it back to distributor, packed, as this is simpler to implement. However, we were striving to achieve faster processing speeds so we recreated the game logic to work on packed bytes. This did prove to be more efficient as bit manipulation has minimal costs since it takes linear time.

Synchronous channels are used for communicating between threads which are particularly useful for handling interrupts as the data remains on the channel until it is required by the distributor. The downfall of this is that each worker will not continue to send its updated section until the previous data has been acknowledged by the distributor.
Using asynchronous channels allows faster workers to be independent of slower workers that take more time to compute the image, avoiding potential bottlenecks. In addition to this, a potential enhancement of processing speed is to send each row of the worker's image to the distributor as soon as it has been updated, instead of sending the whole image at once after modification.

The smallest image, 16 x 16, can be processed by 4 workers in ~43 iterations per second.
The largest image our program can compute, 1024 x 1024 takes ~0.24 rounds per second.