Concurrent Computing - Game of Life
Team 17 Report

Aneesh Anand
MEng Computer Science
aa16169@bristol.ac.uk

Finn Hobson
MEng Computer Science
fh16413@bristol.ac.uk

1 Functionality and Design

1.1 Program Functionality

We started implementing our Game of Life program by changing the contents of the distributor function. Our distributor starts by receiving each pixel of the input PGM image from the channel linked to DataInStream and packs each pixel into bytes in a 2D array. Then the distributor sends eight equal sections of this array to eight concurrently operating worker functions. Once each worker has received its unique section of the image matrix, it begins to calculate the updated pixel values based on the Game of Life transition rules.

We have updated the implementation of the orientation function to pause processing when the board is tilted. When a tilt happens, the orientation function sends a signal along a channel to the distributor, which forwards this signal to each worker. The workers then temporarily stop processing and send the number of live cells in their section of the image matrix to the distributor. The distributor prints out a status report to the console containing the number of rounds processed, the current number of live cells and the processing time elapsed since the image matrix was read in from DataInStream.

We created a new function called buttonListener that reads in values from an input port on the xCORE-200 eXplorerKIT whenever a button is pressed. Before the initial PGM image is read in and processing can begin, the distributor waits for a signal to be sent from buttonListener that the SW1 button has been pressed. Once processing has begun, if the SW2 button is pressed, buttonListener sends a signal to the distributor, which triggers an export of the current image matrix. Each worker sends their section of the image matrix back to the distributor, which sends the current image matrix to DataOutStream. An export can be triggered whilst processing is ongoing or whilst processing is paused and unlimited exports can be triggered whilst the program is executing.

We created a function called showLEDs that lights certain LEDs during the various states of execution. The LEDs are controlled by a 4-bit binary sequence that is sent to an output port on the xCore-200 eXplorerKIT. The distributor sends this sequence along a channel to showLEDs when specific events occur. The sequence is forwarded to the output port and the corresponding LEDs are lit.

1.2 Solving Implementation Problems

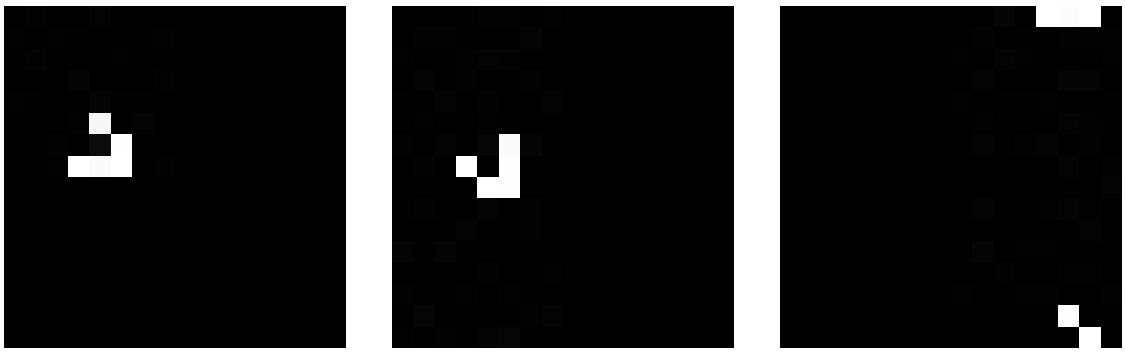
During each processing round, each worker needs an extra line above and below the section of the image matrix that it is operating on in order to calculate the updated pixel values. To increase the efficiency of our program, rather than sending each new section of the image matrix back to the distributor, we created channels between the workers so that they could communicate with each other. This enabled each worker to send and receive the required surrounding lines of their section. In order to prevent a "circular wait" deadlock, we gave each worker an ID number and made the workers with even ID numbers send their lines first while the workers with odd ID numbers receive and then vice versa.

Initially, we stored each pixel as an individual unsigned char (uchar) in a 2D image matrix array, but when we tried to run our program with a 512x512 PGM image our program would not run as memory had overflowed. To solve this problem, before the distributor sends the image matrix to the workers, we packed 8 pixels into each uchar so that the image matrix arrays take up much less memory. The pixels are unpacked each time the image is exported. This has allowed our program to process much larger image inputs.

We added a timer into our program so that we could print the processing time elapsed each time a pause is triggered. However, timers in xC overflow after counting to approximately 42 seconds. To fix this problem, our program constantly checks if the timer has overflowed and increments a variable called timerCount each time it does, so that the printed time can be adjusted accordingly.

2 Tests and Experiments

The following images show the inputs that we used to test our Game of Life program and the output images that the program exported after 2 rounds and 100 rounds of processing. We used all of the images given to us and a 1024x1024 PGM image that we created ourselves.

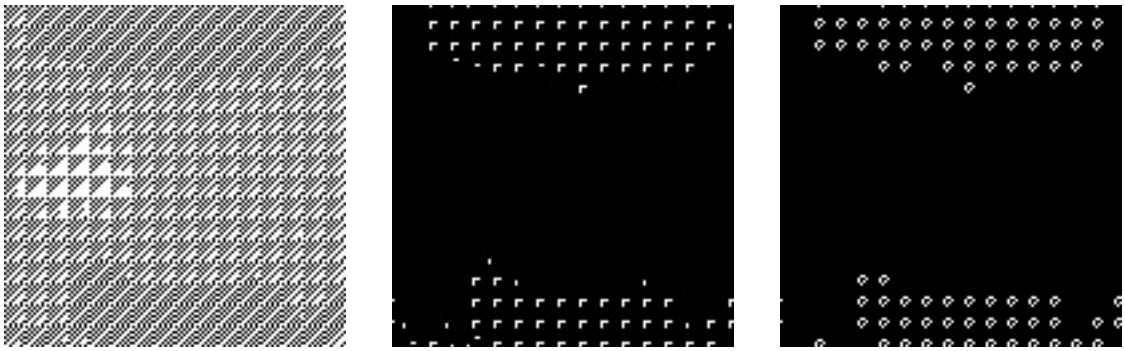


(a) Input (b) Output after 2 Rounds (c) Output after 100 Rounds

Figure 1: Results produced for the given 16x16 PGM image.

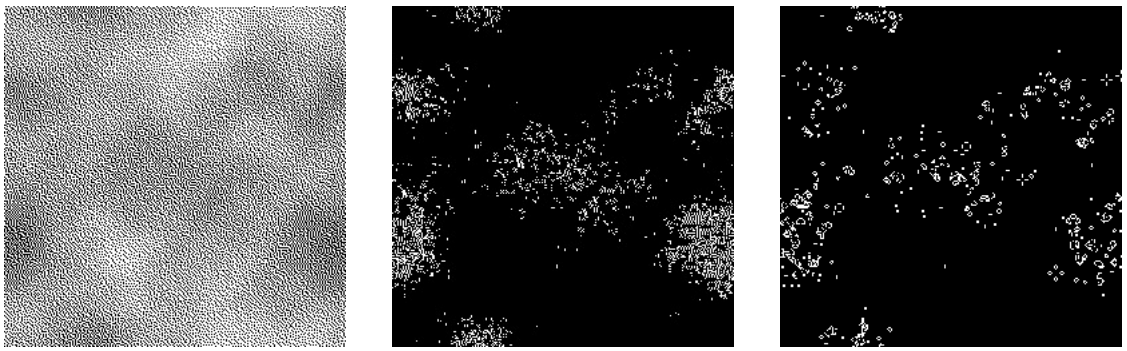
(a) Input (b) Output after 2 Rounds (c) Output after 100 Rounds

Figure 2: Results produced from the given 64x64 PGM image.



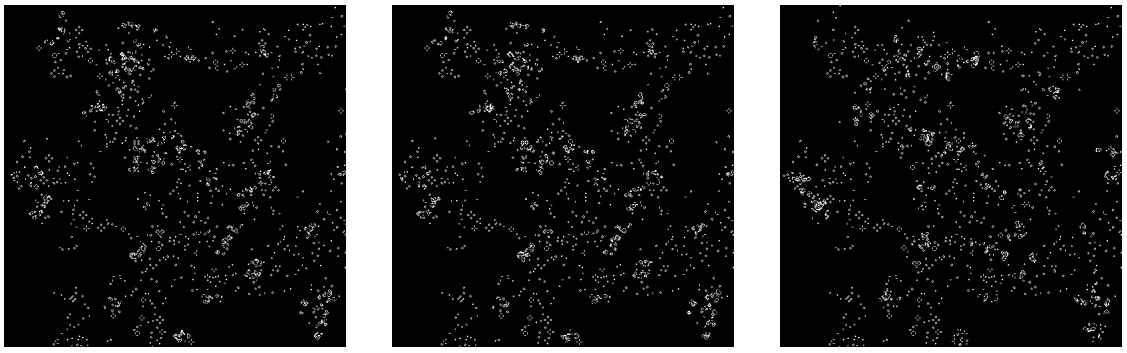
(a) Input (b) Output after 2 Rounds (c) Output after 100 Rounds

Figure 3: Results produced from the given 128x128 PGM image.



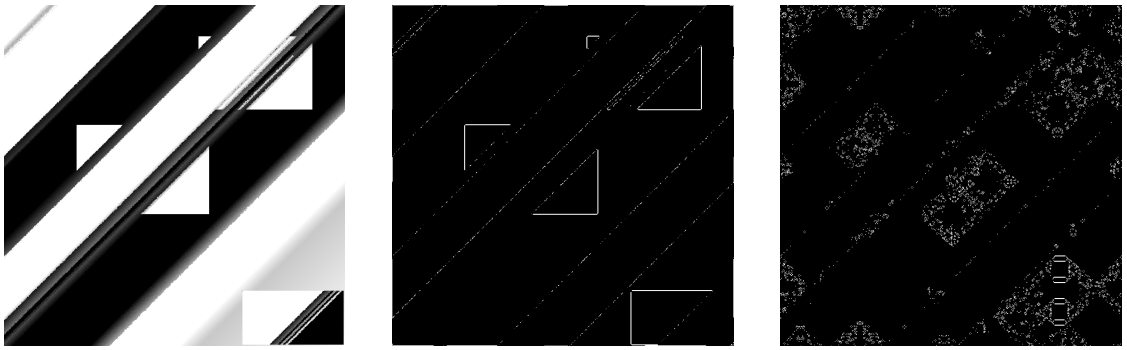
(a) Input (b) Output after 2 Rounds (c) Output after 100 Rounds

Figure 4: Results produced from the given 256x256 PGM image.



(a) Input (b) Output after 2 Rounds (c) Output after 100 Rounds

Figure 5: Results produced from the given 512x512 PGM image.



(a) Input (b) Output after 2 Rounds (c) Output after 100 Rounds

Figure 6: Results produced from our own 1024x1024 PGM image.

3 Critical Analysis

The table below shows various pieces of time data produced by our Game of Life program based on 100 rounds of processing (excluding I/O). We measured the processing speed of our system using the timer features in xC. From the results, our program appears to be very efficient as the smallest three inputs take just over one second to process 100 rounds of the Game of Life. Even for our largest input size (1024x1024) our program calculates over a million new pixel values every round in approximately 306 milliseconds.

Image Size (pixels)	Total Number of Pixels	Time Taken to Process 100 Rounds (ms)	Average Processing Time per Round (ms)	Pixels Processed per ms
16x16	256	1,151	11.5	22.3
64x64	4096	1,151	11.5	356.2
128x128	16,384	1,176	11.8	1388.5
256x256	65,536	4,581	45.8	1430.9
512x512	262,144	18,382	183.8	1426.2
1024x1024	1,048,576	30,600	306.0	3426.7

As you can see from the table, although the processing time per round significantly increases as the image size increases, the pixels processed per millisecond is actually increasing. This is because the speed of the workers calculating new pixel values is much faster than the speed of the communication between workers after new pixel values have been calculated. Therefore when we increase the image size, the ratio of pixel calculations to worker communication is increased so the pixels processed per millisecond increases.

Although our Game of Life program appears to be very efficient, there may have been ways that we could have increased the performance. All of the channels we have used are synchronous channels. If we used asynchronous channels, data from the sender would be dispatched immediately as long as there is space in the channel's buffer, reducing the waiting time in our program. We may have been able to increase the number of workers, reducing the workload for each worker in each processing round and therefore reducing the processing time per round. However, this may cause the cores to run out of memory, threads or chanends. Therefore, we may not be able to increase the number of workers.