Concurrent Report

Ainsley Rutterford
ar16478@my.bristol.ac.uk
Computer Science

Harry Waugh
hw16470@my.bristol.ac.uk
Computer Science

November 30, 2017

1 Functionality and Design

Our system currently uses up to 8 workers to evolve the Game-of-Life repeatedly. Our system is deadlock-free, implements the correct button, board orientation, and LED behaviour, and can process images up to 1264x1264 pixels using memory on both tiles.

The biggest problem we encountered was trying to run images bigger than 512x512. Originally we were reading the bits from the .pgm file into a 2 dimensional array of `unsigned chars`, before 'packing' this array into an array of unsigned chars that represented each pixel by a bit instead of a byte. This packing process allowed us to save space as we only had to store an array which was 1/8th the size of the original array. We proccessed the Game-of-Life on this array. Eventually we had to free even more space in order to run larger images. To do this, we changed the `dataInStream` function, so that it read the pixel values straight into the smaller array, representing each pixel as a single bit. At this point we could process images up to 688x688. We then then realised that in `dataInStream`, we didn't have to store an `unsigned char` array at all. We simply processed each byte and sent each byte to the distributer as we read from the file. This meant that instead of storing an entire array of `unsigned chars` of size `[ImageHeight][ImageWidth / 8]`, we stored a single `unsigned char`.

We also encountered problems while implementing a timer. The first problem was that the timer seemed to overflow roughly every 42 seconds. We decided to create a function that would compare two times given, and would return whether or not the timer has overflowed. We would then call this function when the board was tilted, and if the function returned true, we would increment a counter which counted how many times the timer has overflowed. The time displayed would be the timers current value added to the overflow value multiplied by the counter value. Using this method, our clock no longer overflowed.

Early on in the development of our system, our workers functioned differently. Each worker would work on a single byte of data. We would send each worker a 3x3 array of `unsigned chars`, with the middle char being the char that the worker would work on. The worker would then send only the completed char back. We realised that this was very inefficient as each worker required 8 extra bytes to work on a single byte. Image processing was very slow using this method so we decided to update our workers to work on a whole strip of an image at once. This meant that for whole chunk of an image to be worked on, only an extra strip of pixels at the top and bottom of the strip had to be sent as the sides of the image wrap round to each other. Using this new worker strategy, our image processing was up to 11 times faster.

2 Tests and Experiments

For each image provided to us, we will show the image after 1, 2 and 100 iterations respectively. We will also include randomly generated 1024x1024 and 1264x1264 images and their subsequent iterations. The results are shown in Figures 1-7.

The biggest virtue of our system is the size of images that we can process succesfully. We spent a long time redesigning the way our `worker, dataInStream, dataOutStream, distributer,` and `gameOfLife` functions worked. By only storing one 'packed' array of bytes in the `distributer` and only the previous and current line to be processed in `gameOfLife`, and only storing one temporary `unsigned char` in `dataInStream`, we managed to process images up to 1264x1264.

We carried out experiments comparing the time taken for an image to be processed when using different workers. We experimented with using 1, 2, 4, and 8 workers. The results are show in Figures 8 and 9. We generated the first graph by plotting the image size against the time in seconds. Each line represents the time taken for each worker to generate 100 iterations of that image. The time taken for 100 iterations of the Game-of-Life to be processed for each image size is provided in the table shown at the bottom of page 2. This also includes the number of workers used.

One limitation of our system is that it can only work on images that have a width that is divisible by 8. This is due to the fact that we 'pack' 8 pixels into an `unsigned char` to represent each pixel by a single bit. Another limitation of our system is the size of images it can process, due to the limited amount of memory provided by the board. In order to process images any larger than 1272x1272, we would have to either, attach another board to our own, or remove the functionality of the buttons, LEDs and orientation sensors of the board.

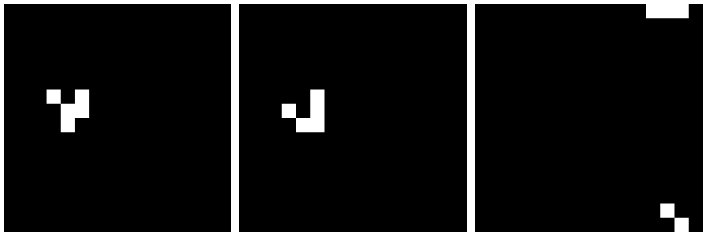

Figure 1: 16x16 images.

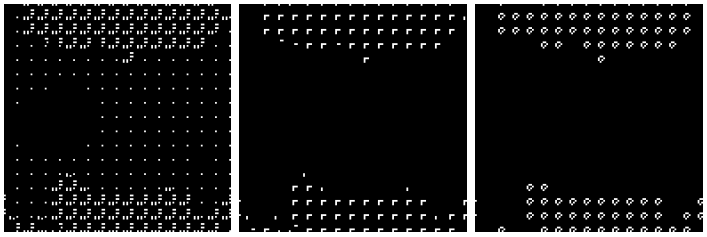

Figure 2: 128x128 images.

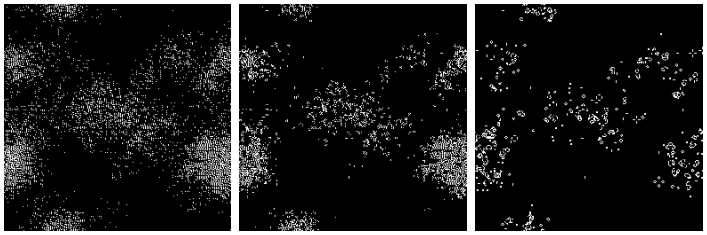

Figure 3: 256x256 images.

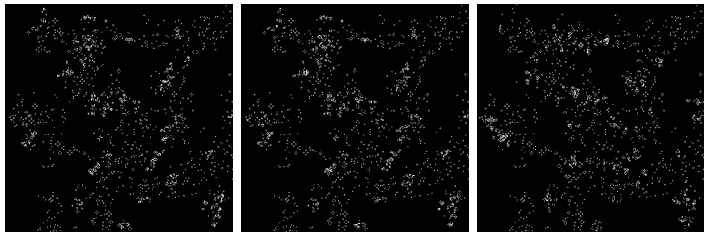

Figure 4: 512x512 images.

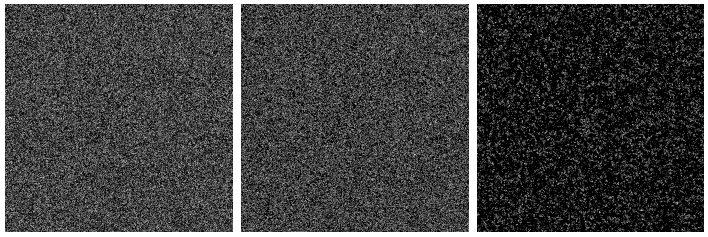

Figure 5: 1024x1024 images.

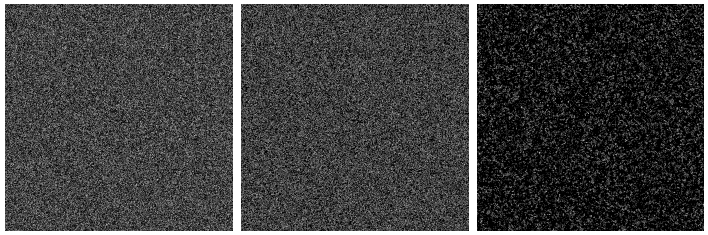

Figure 6: 1264x1264 images.

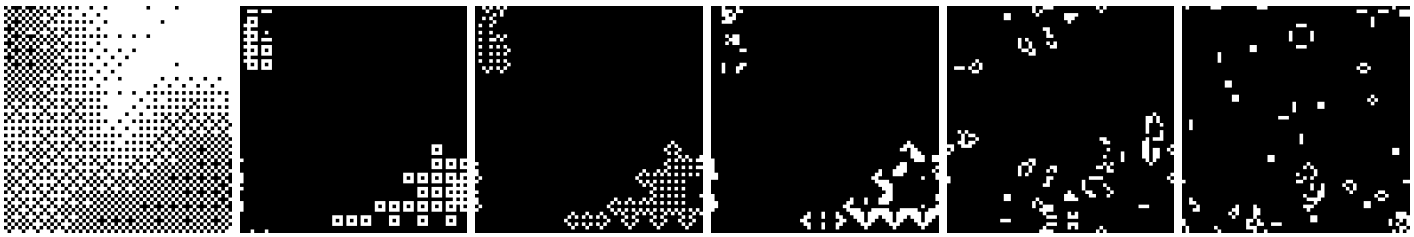

Figure 7: 64x64 image after 0, 1, 2, 3, 100, and 1000 iterations respectively.

	1	2	4	8
16x16	0.27	0.16	0.11	0.10
64x64	3.50	1.82	1.00	0.81
128x128	13.40	6.81	3.54	2.83
256x256	52.48	26.76	13.76	11.09
512x512	212.43	107.74	55.50	43.65
1024x1024	849.73	431.77	220.34	174.81
1264x1264	1408.64	717.47	370.84	294.24

Time taken in seconds for the image size specified on the left column
to be processed by the number of workers specified on the top row.

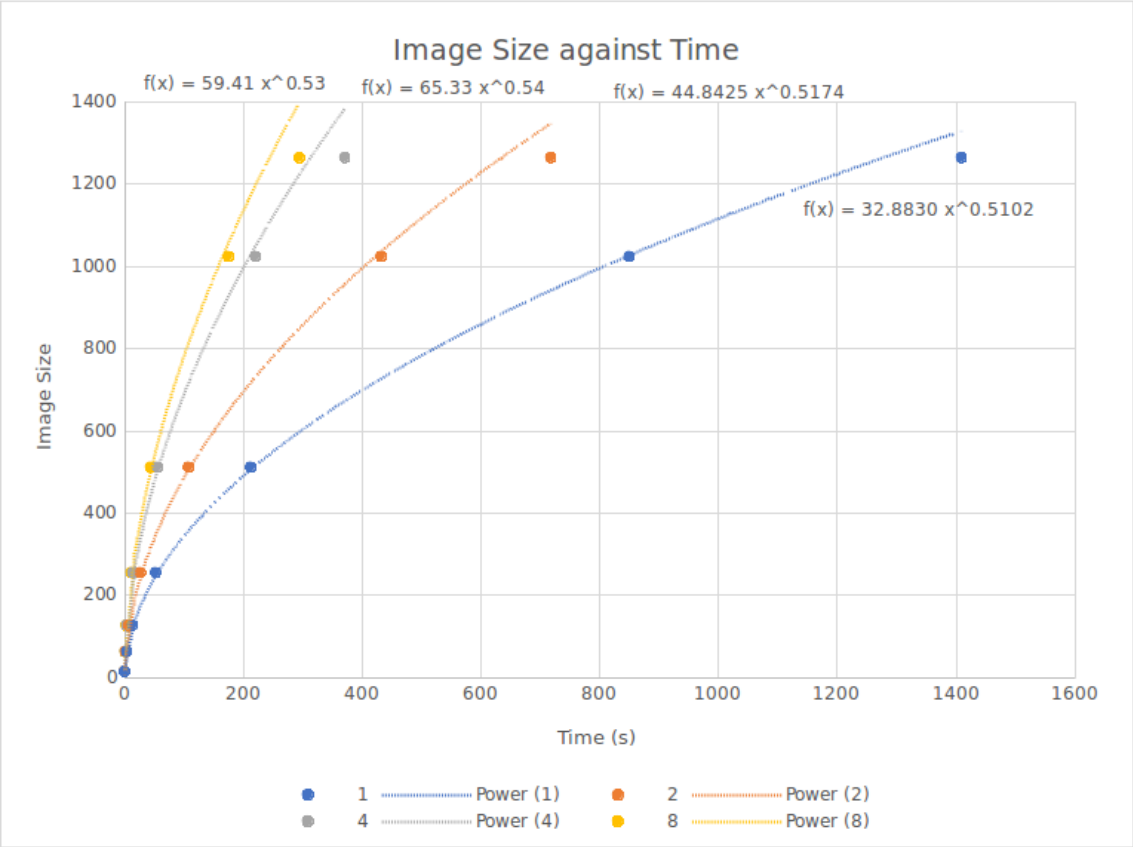

Figure 8: Image Size against Time.

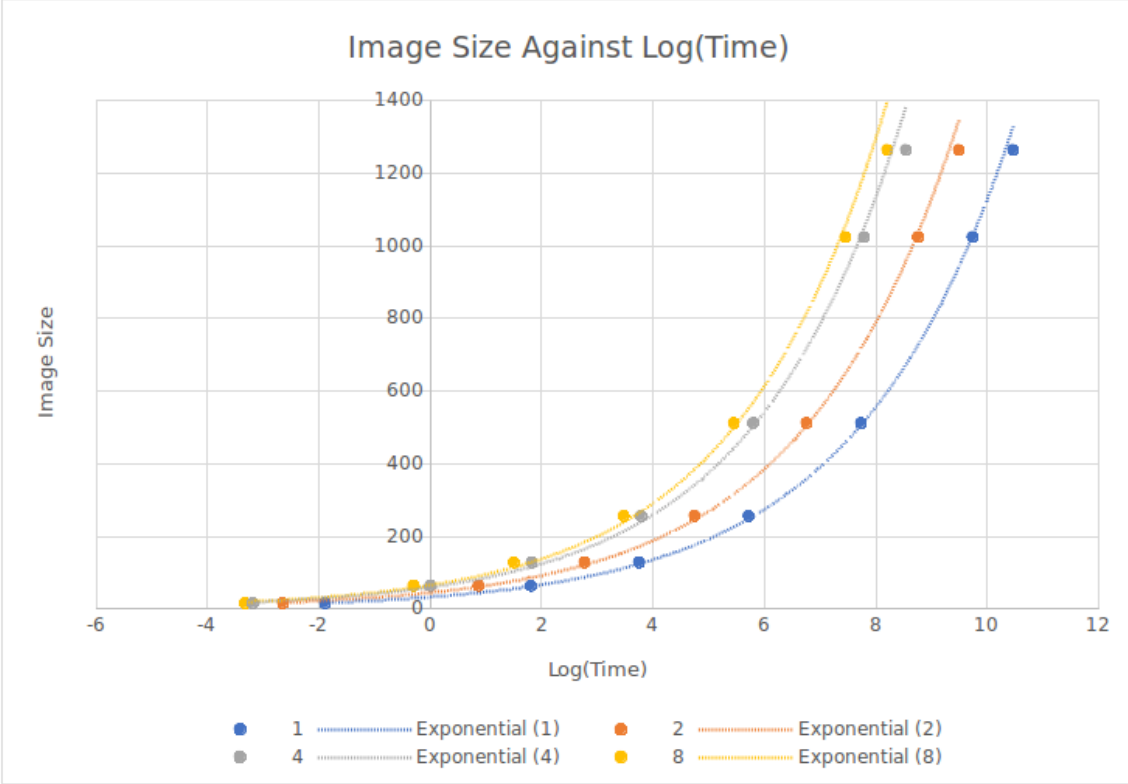

Figure 9: Image Size Against Log(Time).

3 Critical Analysis

The maximum size our system can process is 1264x1264. Our system can process these large images quickly, for example our program processes a 1264x1264 image in roughly 2.9 seconds. Our system also does well at processing small images, processing 100 iterations of a 16x16 image in 0.10 seconds.

Looking at the table at the bottom of page 2 shows the time taken for 100 iterations of the Game-of-Life to be processed using different numbers of workers. Initially, when looking at our table, we noticed that there seemed to be diminishing returns, especially when increasing from 4 workers to 8 workers. For example, when processing 100 iterations of a 512x512 image, there is a 49.3% decrease in time taken when increasing from 1 to 2 workers. Similarly, there is a 48.5% decrease in time when increasing from 2 to 4 workers. However, when increasing from 4 to 8 workers, there was only a 21.3% decrease in time taken. This can be seen visually when looking at the graph in Figure 9, where the gap between the lines are significantly closer for 4 and 8 workers. This graphs X axis is the logarithm base 2 of the time taken.

From the data and the graph in Figure 8, we generated a trendline and its equation which allows us to estimate the processing time needed to generate various image sizes with different numbers of workers.

We noticed that reading and writing images was very slow initially. We sped this up slightly by sending a 'packed' array to the distributer instead of an array representing each cell by a byte. This meant that we were sending a set of values that was 1/8th the size of the set of values that was originally being sent. However, reading and writing is still extremely slow. If we had more time for improving the system, perhaps this could be improved by reading in multiple lines from the file at a time. Though our system does not deadlock over thousands of iterations, and we have tested it extensively manually, an improvement would be to implement these tests as automatic testing. Overall, we think that our system performs to the the specification, handling images well beyond 512x512 in a reasonable time frame.