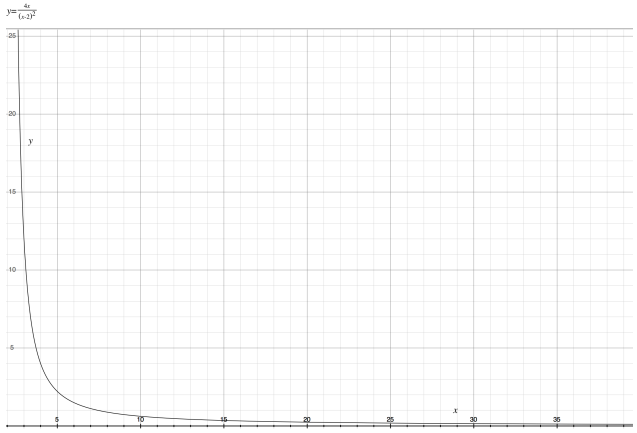
Oliver Goldstein & Stanley Green - Computer Science (Both on BSC)
og14775@my.bristol.ac.uk & sg14776@my.bristol.ac.uk

Report : XC Implementation Of Game Of Life og14775/sg14776

Theoretical analysis of the method we chose.

In Year 1, Computer Architecture recommended that as the size of processors and wires decrease, resistivity of those same wires increases, increasing heat and limiting information flow through networks. What was also mentioned was the routing of data through these networks. Often, such data cannot be directly sent to where it is needed, resulting in the use of indirect routing - which represents a cost in time. Understanding these abstract principles, we thought of various methods to implement The Game Of Life.



The first implementation we considered, every round, the distributor sends the image dissected in a predefined way, to each of the workers. These workers would then process the image and send it back to the distributor, which would then re-send it to all of the workers. We saw this as a facile solution to the communication of ghost rows (the edge rows that are needed to process the outermost layer of an image) between workers. The distributor would send the image along with the ghost rows. We thought about this and decided it was a poor solution for primarily two reasons. On small images with many workers, for every pixel we work on, we are sending at most eight neighbours along with it, solely to process that single pixel. As the size of the image portion that each worker receives, increases, the ghost rows represent an ever-diminishing proportion of the processing and thus amortise in utility, which is depicted in the right hand graph which assumes an image of square size. We have four rows to send, two being of length width and two being of length height. We work on the inner part of the image, which is (height-2)*(width-2) in size. For example, with an image of size 20 by 20, the ghost rows would represent proportionally 10 times less than with size 5x5. The graph does at some point tend to infinite, unlike reality, but the point still stands. As well as this we found that given a single distributor, it would have to read in the data sequentially from each worker. This would act as a bottleneck on the processing, assuming the workers blocked on sending data, effectively rendering the program sequential. In order to test this, we timed a round trip of splitting and sending a 512 by 512 image to 12 workers. We then compared this to the time it took to process the image, by timing the processing of an image by 12 workers and taking the longest time. We found that communication to 12 workers by splitting a 512 by 512 image between 12 took 927183 ticks but the time for the processing by the longest worker on that round took 4809002 ticks. We see that the processing is around 5 times longer. The communication, as it's sequential would have taken O(mn) where m is the number of workers and n is the size of the array to transfer. Both the time complexity and the space complexity are far worse than the solution we implemented. The distributor would store both the processed array and the unprocessed array (removing the need by streaming would be difficult as the ghost rows need to be sent - this is not a contiguous flow of data). Each of the workers would store the processed and unprocessed image split. Overall this means that four images are stored in the worker and the distributor combined.

Our implementation has a space complexity of only two images stored, one representing the unprocessed image and one representing the processed image. The distributor sequentially streams these immediately to the workers. At the beginning of each round, the workers are told whether to do a data out event or to pause due to an accelerometer event before communicating their ghost rows. The distributor divides up the image into rectangular blocks, where the width is the width of the divided image and the height is the total image height divided by the number of workers. The last worker will take the remainder of the division and thus may have a marginally different height (the difference upper bounded by the number of workers) to other workers. This enables us to have both even and odd sized images of any multiple.

Each worker communicates its ghost rows in a chain. One worker is allocated to send its top row before receiving its bottom row and the rest first receive their bottom row and then send their top rows. This then repeats except with the converse row in the opposite direction. Upon processing the image the worker in O(1) time changes the reference to the old array and new array by having a variable that increments modulo 2. The value of this flips between 0 and 1 and is used to index the old array and the new array in the worker. This is far faster than alternatives such as memcpy which operate in O(n) time. The workers having only a top and bottom channel is optimal, as we do not exceed the channel maximum on either tile.

In addition to the above we have managed to implement a number of optimisations. Firstly, we have conserved space by a factor of 8 by using bit packing. We do this by implementing a one dimensional unsigned char array, with get and set bit operations, using a single bit to represent either white or black. This has allowed us to implement images of up to approximately 1.7MB. In addition to this, we have exploited the use of preprocessor directives to dynamically link workers at compile time. We simply modify a single number in a header file and all channels are linked. Furthermore the array size that each worker receives is dynamically calculated at compile time. We also use streaming channels, which reserve a physical wire and have a larger buffer, for our inter worker communication. As well as this our workers are roughly evenly distributed on each tile meaning that on an image of 1265 by 1265 we use 262104/262144 on tile 1 and 236076/262144 on tile 0.

We have considered a number of different approaches to avoid deadlock and livelock, using small of CSP to test our code in the early stages. As the state space of this code exploded it became harder to analyse using basic CSP. We wanted to ensure that there were no or at least minimal critical sections located within the code and that any button presses or tilts were lossless events. To do this we had a dedicated core listening for events for the accelerometer and a core listening data in/out events. We were aware that as there are multiple workers, they could exist at multiple points in code at any one time. The way we avoided this is as follows. The first worker to reach the synchronisation point asks the distributor if it is the first worker. The distributor then enters a case statement having received input from its channel array to the workers. It then waits for all other workers to ask if they are the first worker or not. Whether the worker is first or not will dictate whether it sends or receives its top or bottom rows first. After the distributor tells the workers whether they are first or not, the worker asks the distributor what event it should do. It has a choice of either a data out event or a pause event. This relationship, whereby the worker every time, asks the distributor what it should perform avoids different workers catching on different case statements if they reach different points in code at different times, they will all receive a uniform answer by the distributor at any given round. At the very end of the synchronisation point the distributor waits for the last worker to finish. It then sequentially re-synchronises with all the workers. After this point all the workers continue processing. No matter if one worker was particularly fast at processing and reaches the synchronisation point again, no error will occur as after the re-

synchronisation point, no more channel communication between the distributor and worker occurs. Furthermore, the select statement has the [[ordered]] attribute applied to it, meaning the workers will not block the accelerometer or data in/out case statements from activating.

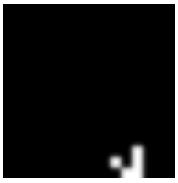
Testing and Timing

In order to test the images yourself, in the PGmIO.h files please edit IMWidth and IMHeight and the numberOfWorkers.

So far above we have tested the processing versus communication time. We saw that a round trip of communication was 1/5th of the time taken for a worker to process a 512 by 512 image. We also tested the difference between streaming asynchronous channels and synchronous channels for doing a input and output. Although faster the streaming channels were only 99.0% of the speed of non-streaming channels (1486149349/1494972227 ticks). We suspect this difference lies in the fact that the bottleneck must be in the _readinline and perhaps the fread method, rather than the communication between the distributor and each of the workers, or the fact that streaming channels are only marginally useful.

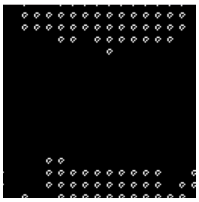
The images generated at round 31 are all below.

16 by 16

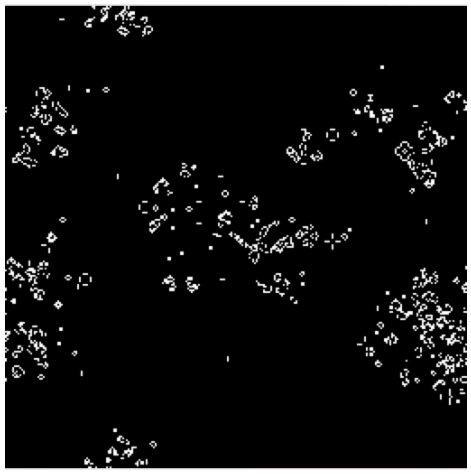


64 by 64

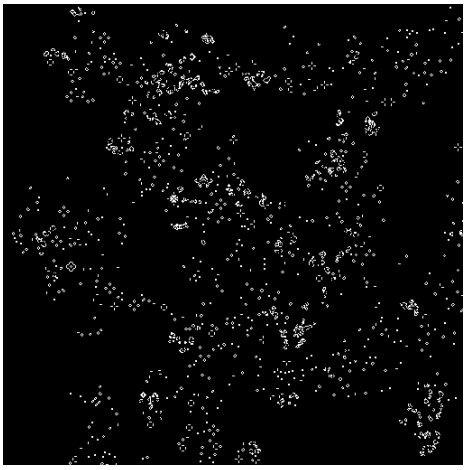
128 by 128



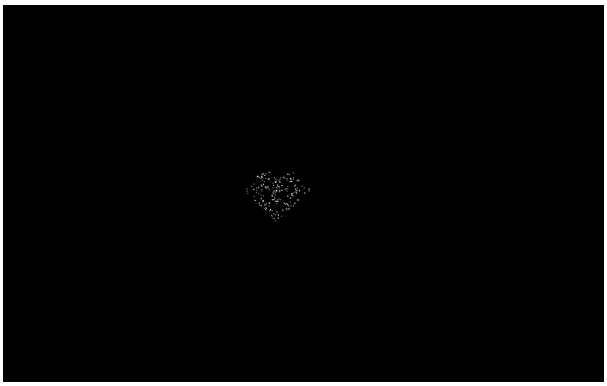
256 by 256



512 by 512

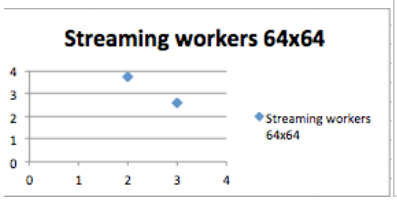


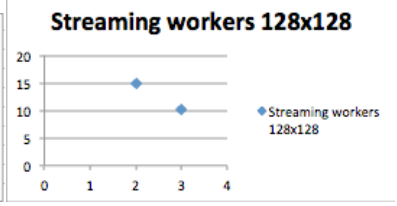
The following is width:1300 by height:1273 image processed after 30 rounds. This is only works with 10 workers due to the uneven distribution of workers around the tile. This approximates the maximum image size our system can take. This demonstrates the merits of our system as the bit packing means we can store up to 8 times as much as we otherwise could. Whatsmore because we allocate a last worker to take the remainder using modulo arithmetic, we can use different height images of awkward sizes without any kind of compromise



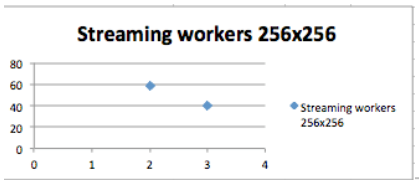
Oliver Goldstein & Stanley Green - Computer Science (Both on BSC)
og14775@my.bristol.ac.uk & sg14776@my.bristol.ac.uk

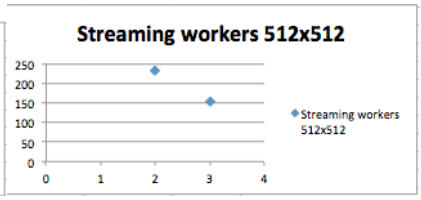
In each of the graphs below the y axis represents the time and the x axis represents the number of workers used.

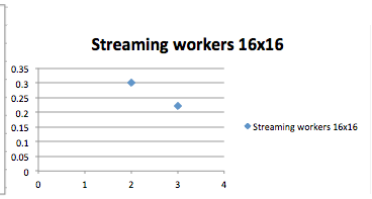


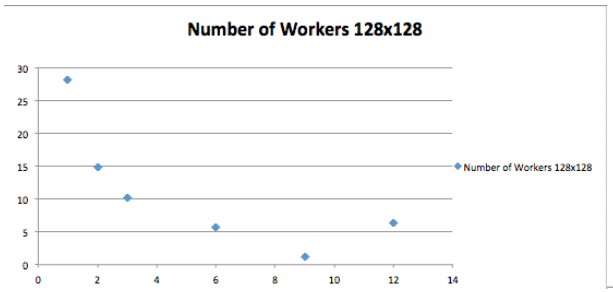


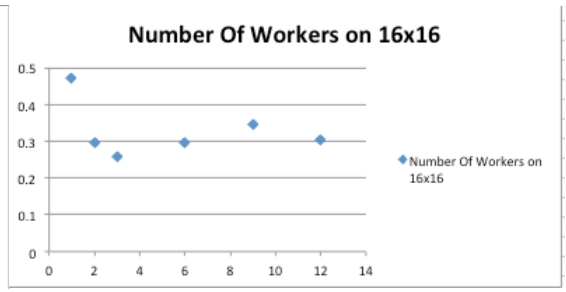
RESULTS

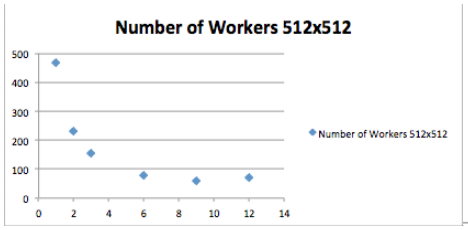


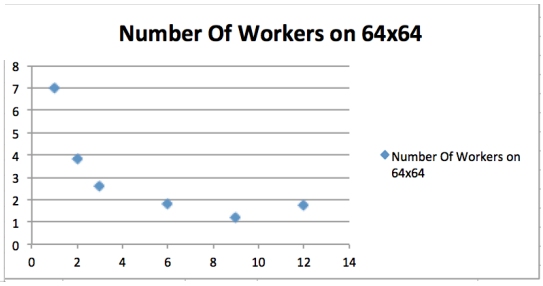


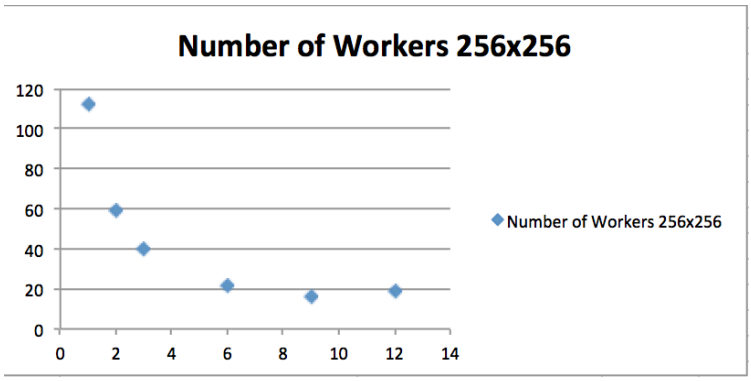












Number of Workers 512x512	Time (3.s.f) 50 rounds
1	467
2 streaming	231
2	231
3 streaming	155
3	154
6	79.9
9	60.1
12	71.3

Number of Workers 128x128	Time (3 s.f.)
1	28.2
2 streaming	14.8
2	14.8
3 streaming	10.2
3	10.2
6	5.66
9	1.2
12	6.32

Number of Workers 256x256	Time (3 s.f.) 100 rounds
1	112
2 streaming	59.3
2	58.2
3 streaming	39.9
3	40.3
6	21.7
9	16.2
12	19

Oliver Goldstein & Stanley Green - Computer Science (Both on BSC)
og14775@my.bristol.ac.uk & sg14776@my.bristol.ac.uk

Number Of Workers on 16x16	Time (3 s.f.)		Number Of Workers on 64x64	Time (3 s.f.)
1	0.472		1	6.98
2 streaming	0.296		2 streaming	3.8
2	0.302		2	3.72
3 streaming	0.258		3 streaming	2.62
3	0.222		3	2.61
6	0.295		6	1.81
9	0.345		9	1.2
12	0.302		12	1.77

Timer test results

Sequential v concurrent

Across all tested image sizes, 1 worker had significantly slower times than multiple workers, compared to 2 workers. For example, on a 16x16 image 1 worker took 0.472 seconds to complete 800 rounds, compared to the 0.296 seconds required by 2 workers to complete the same number of rounds, one worker took 159% longer than 2 workers. On large images such as the 512x512 1 worker took 200% longer, with a best case of 9 workers one worker took 770% longer to process the same number of rounds.
The increase in speed observed by adding more workers in comparison to a single worker showed a general trend of increased speed up to 9 workers, and then began to rise as more workers were added, with 12 workers typically being slower than 9 workers, as showed in the graphs, with the exception of the 16x16 image. These results clearly demonstrate that using multiple workers was in all cases faster than using a single worker, showing that the concurrent processes are superior to sequential processing of the image. 12 workers have slower communication because they need to transfer more data relative to the amount of data being processed as discussed above. One worker is slower because it has to do everything alone even though its haves on communication highlighting the benefits of our system. The trend is that the fastest results occur with 9 workers. We would expect with larger images the 12th worker amortizes and brings more benefit as there is less relative data transmission.

Synchronous v asynchronous(streaming) channels

Replacing synchronous channels with streaming channels made no significant difference to the speed of processing in our system. Across all image sizes, results for programs with the same number of workers that had either streaming or synchronous channels were indistinguishable.
For an image size of 64x64, using 3 workers with synchronous channels produces a result of 2.62 seconds, the same number of workers with streaming channels produces 2.61 seconds. No image size with the channels switched between these two types displayed any significant difference to the speed of the program.
A streaming channel will provide the highest data rates between threads, but can only read when the buffer is not full. There are bottlenecks in the workers case statement that are not affected by the increased data rate of the streaming channel.

Critical Analysis & Evaluation

There were many different things we could have improved. We considered whether we should have identified black areas of the image and should not have worked on them, however in order to check that there would have been a significant slow down across all of the processing rounds.

We have identified the bottleneck in our code as being the communication of rows between our workers.
As we have seen, the workers exchange their rows in a sequential manner; having one channel between the workers means it is imperative that they do not both send across the same channel at the same time. However, this results in a slower processing capacity, the exchange has to happen at the start of every round, and for each round 2n rows, where n is the number of workers, must be sent one after the other, giving a time complexity of O(n).

If these rows can be exchanged in parallel, the time each round takes to complete could be reduced significantly.
One idea to allow more than one worker to send rows at the same time would be to use the distributor to tell workers that are not connected to each other to pass their rows. For example, in a chain of 10 workers, the 1st, 3rd , 5th, 7th and 9th workers could send their rows to the worker above or below them (worker 1 pass top row to worker 2, 3 to 4, etc). Then the 2nd, 4th, 6th, 8th and 10thworkers could be told to send their rows to the worker above (i.e. 2 to 3, and 10 to 1). This would reduce the time required to send the rows by a factor of n/2, in the example of 10 workers, it would reduce the sending of 20 rows one after the other, to 4 sets of 5 (n/2 = 10/2 = 5).

Another addition would have been the implementation of the HashLife algorithm, which exploits spatial and temporal redundancy in the Game Of Life rules. Examples of this include still life patterns (stationary patterns) and oscillators, which are repeating patterns. These patterns can be memoized.

We could have connected two of the boards as well and had access to double the memory i.e. 2 times 512kb and twice the number of cores. With more time we could have performed a greater number of statistical tests, for example t testing for significance to see if increasing the number of workers actually has a statistically significant effect.

Space complexity could have been improved by replacing the system of using 2 arrays, one new and old array, with a line buffer of up to a few lines long. As we read from the unprocessed image we transfer into a line buffer. Once we have processed the first few lines, we no longer need the lines above and so can steadily transfer from the line buffer back to the unprocessed image. We would have to reserve the top and bottom lines however as they are needed at certain points in the code. We would only then have to store one array.