Bittor Alana
lo18144@bristol.ac.uk
Math & Computer Science (Erasmus)

Kshitij Upmanyu
ku17217@bristol.ac.uk
Computer Science

1 Functionality and design

When tackling the implementation of the Game of Life, we have encountered different challenges to face, in terms of running parts of the code concurrently:

- Amount of memory allocated vs. thread communication: the amount of data that each thread has access to is a key decision on the performance of the programme. We can either chose to overlap information, in such a way that the different threads won't have to communicate with each other (but duplicate data), or we can decide to strictly give them different data which do not overlap, and have them interact with each other.
- Another issue linked to the one above mentioned, is what parts of the processing we assign to the working thread. As we will explain in further detail afterwards, this may make the performance vary. We will discuss two major variants, one which will run two concurrent processes first and save some work for the workers afterwards, and another one which will set the workers to run from the very beginning of the processing.

One of the problems encountered when coding a solution was to make an appropriate use of memory. For that end, we have decided to model the images as matrices of characters, where each character, which is a byte, contains the information of 8 consecutive pixels of a row in the picture. Thus every pixel uses only a bit of memory. We also tried using 16-bit integers (for some reason they would only work with 16). This version was a bit faster but less memory efficient, so we mainly stuck with the byte representation.

The main variant, or **Version 1**, runs three threads at first. One of them will visit each pixel. If the pixel is alive, it will tell another thread to increase the number of live neighbours of that pixel's neighbours by one. What pixels are alive will be communicated through a buffer. This self-implemented buffer is a form of asynchronous communication, whereas most of our other threads use synchronous channels. After the number of live neighbours of each pixel are counted, we run the worker threads. Each of them takes strictly the proportion of image that corresponds to them, and as it already knows the number of living neighbours, really quickly updates the image.

If we make the worker threads count living neighbours as well, we need either communication between blocks (**Version 2**), or overlapping of data (**Version 3**). What differences these different approaches have in terms of performance will be discussed in Section 2.

2 Tests and experiments

The main version of the game, in which we only run the workers after counting neighbours, is very speed-efficient but not so efficient memorywise.

The second version, which sets the workers to work from the beginning and make them communicate, is less efficient in terms of speed, but can handle bigger images.

The third version, which an extra two lines to each worker so that they don't have to communicate, is somewhere in between the other two versions, having a relatively bad memory efficiency but quite speedy.

The table below shows the average speeds for the images given, in rounds per second, taken with high amounts of rounds (we must take into account that different moments of the game, with more or less living cells, can result in different speeds as well. For these calculations, we used 8 worker threads.

	16x16	**64x64**	**128x128**	**256x256**	**512x512**
1st version	87.17 r/s	89.82 r/s	52.13 r/s	12.85 r/s	3.58 r/s
2nd version	91.13 r/s	89.32 r/s	25.1 r/s	6.23 r/s	1.54 r/s
3rd version	90.74 r/s	90.41 r/s	26.85 r/s	6.78 r/s	1.73 r/s

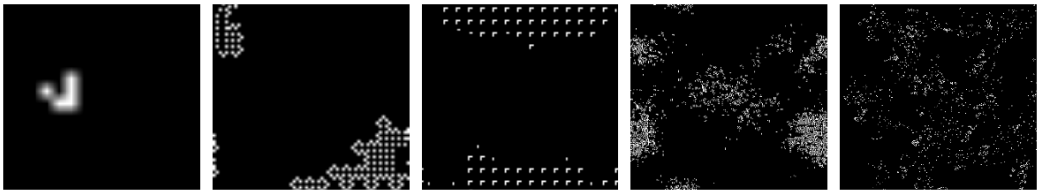

Figure 1: Results drawn from each test picture after two rounds

There is a huge amount of little variations that can be made to each version. A choice that affects performance is how we encode the information of the picture. The code of the first version includes a definition of a type and an amount of bits at the beginning. Thus we can easily switch from representing the picture as a matrix of characters (of a byte each) to representing it as a matrix of 16-bit integers. Both cases produce the same outputs, but the performance is different: for the given 256x256 image, for example, the integer version reached 100 rounds in 6.64 seconds, whereas the character version took 7.78 seconds to do the same. However, the integer version is worse memorywise, and cannot handle 512x512 images.

Lots of variations can also be introduced to the second version. When two consecutive blocks are communicating, they can either communicate very often, asking for a small piece of data (three bytes, for example) at a time, or communicate just once, giving each other the whole line of the edge.

Our 2nd version, which uses the frequent communication, is very slow as we see on the table. Thus, we alter it so that each block sends and receives its edge lines at once, and then processes it. This yields the following speeds:

	16x16	**64x64**	**128x128**	**256x256**	**512x512**
2nd version, variant 2	90.85 r/s	90.11 r/s	24.3 r/s	6.23 r/s	1.55 r/s

As we see on the table, this version is by far the slowest one.

We generate a random 768x768 image, and all versions are able to handle it.
Now let's move on to one of the major variations that we haven't discussed yet: the number of worker threads. All the above tests were carried out with 8 blocks, or 8 workers. What if we try with 4? The results are the following, taking averages after around 30 seconds:

	16x16	**64x64**	**128x128**	**256x256**	**512x512**
1st version	90.85 r/s	90.81 r/s	43.34 r/s	11.34 r/s	MEM FAIL
2nd version	90.90 r/s	90.86 r/s	21.73 r/s	5.51 r/s	1.46 r/s
3rd version	91 r/s	90.11 r/s	21.57 r/s	5.54 r/s	1.36 r/s

We see that generally, everything slows down, and it's also less memory-efficient (**V1** cannot handle 512 anymore), therefore the 8-worker version is better.

We have tried our own pictures, also of bigger sizes, one of them inspired by 'Pulsar'. When running a 768x768 'Pulsar' image on our **Version 2**, it did 71 rounds in 103.464 seconds, which averages 0.686 r/s. We also made a 896x896 which just consisted of several copies of one of the repeating figures in 16x16, and it also worked. It did 82 iterations in 163.501 seconds, averaging 0.5015 r/s. A small version of the pulsar lets us see how it behaves cyclically:

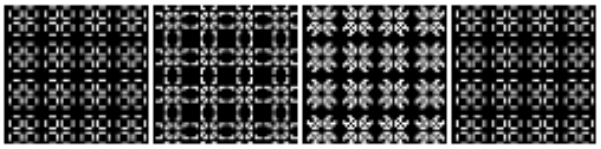

Figure 2: Original image and three iterations (note that it is 3-periodic)

Note that for this sizes the input and output of the data takes a huge amount of time.

Doing many tests, we deduced that **the biggest image size** that our system can handle is 960x960, but it cannot handle 1024x1024 images. Surprisingly, this result was achieved in **V3**.

	Speed	**Memory use**
V1	The quickest one by far	The poorest, cannot handle 768x768 images
V2	The slowest one with 8 threads	Quite efficient, can handle up to 944x944
V3	Quicker than **V2** with 8 threads	Can handle up to 960x960

3 Critical analysis

We are quite happy with the different range of results we have been able to obtain. However, we are also very surprised by the fact that our **V3** outdid **V2**.

V1 showed an impressive speed for the Game of Life, developing the game at a pace of almost 13 rounds per second for 256x256 images. This is because it saved up a lot of work of neighbour-checking, as once it knew a cell was dead it just skipped it for the rest of the process. However, it lacked memory efficiency, and it couldn't manage big images, as it used a matrix of counters of neighbours, which is four times as big as the bit-packed picture itself.

V2 was the version we hoped would be the most memory-efficient. However, this turned not to be true. It can still develop the game very rapidly and with pictures of very big sizes. However, we may be using too much communication and too many variables, as other versions outdid this one in speed and memory-efficiency. We are aware that the amount of communication could have been decreased if we only asked the other blocks for one bit per bit of the edges (we took 3 each time, but 2 of those 3 are reused by the next bit). This has probably meant too much message-passing and perhaps cost us some efficiency in both speed and memory. This is clearly one improvement that could be made.

V3, which we expected to be an intermediate of the two versions in terms of both memory and speed, proved us wrong in one regard, as it turned to be the most memory-efficient. However, we did correctly predict it to be an intermediate in speed. It is still quite speedy, and it is impressive the size of the images that it can handle.

Very surprisingly, **V3** was the one which was able to process the biggest picture, 960x960 being that figure.