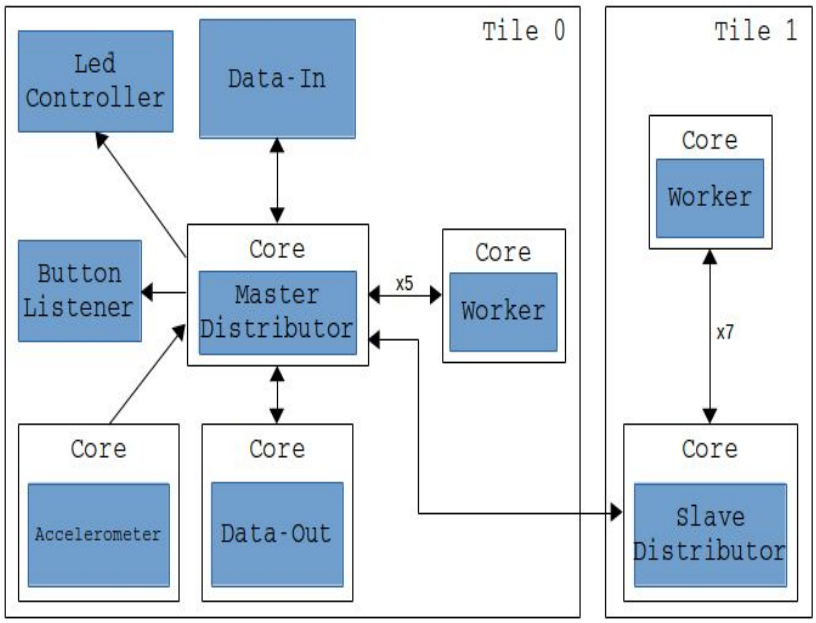
Andreas Georgiou ag14774
Constantinos Theophilou ct14872

Cellular Automaton Farm Report

Functionality and design:



The diagram on the left is a visualisation of each core's job in the Cellular automaton implementation. In the first tile, we have 5 workers, 1 data in stream, 1 data out stream and finally the (Master) distributor. In the second tile we have 1 (Slave) distributor and 7 workers. This way, two star networks are created with the distributors connected to each other, which in turn connects the two tiles and synchronizes the system.

Image reading: To be able to process large images, each byte of memory represents 8 cells. Since XC does not provide such a data structure, we created our own library(bitArray.h) which provides the functions required to set, clear, change, and read bits from an one-dimensional array of unsigned characters. The library makes sure that each line starts from a "fresh" byte, rather than the middle of the previous byte. This significantly increases performance during communications between threads, since it allows for 8 bits at a time to be transferred. That would be otherwise impossible, because the system would be unable to calculate the exact beginning of each line. Also, we included dynamic read-in of the image, meaning the system can distinguish the size of the image during runtime.
As a team, we decided that it would best be fitted to use interfaces instead of channels, to assure communication between individual threads and exchange data between arrays using memcpy. Since, at least a whole line has to be sent, bitArray.h provides a function, that, given the number of lines and the width of each one, it can calculate the exact number of bytes that should be copied.
Interfaces: Once we started plotting our design, we figured out that threads such as the button listener, LED controls and data in stream, would waste cores since they are not used continuously. For the sake of concurrency, we decided that, the best solution to this, was making the processes, such as the three stated above, [[distributable]] which would in turn mean that such a system would be in need of interfaces, so as to save cores from being wasted. To make this possible, all those threads had to accept requests rather than sending raw data once they received it. For example, instead of sending a button press event from the button listener, to the relevant thread, the thread itself requests a specific button press and then that thread is responsible for listening in case of that event.
Notifications: To avoid unnecessary identical loops on both sending and receiving threads, we made use of notification functions and thus the system is more event-driven because the workers do not have to poll the distributor for data. An example of that is, the use of notifications between the distributor and workers. The distributor receives the image and if it

has available lines, it sends notifications to the workers saying that it has data. The workers, receive the data, process the lines and once finished, return the lines to the distributor, so they can, then get more data.

Data Distribution: Data distribution between tiles and cores was a big issue to our implementation. We wanted the image to be fairly divided so that all the workers had the same amount of data to process. In terms of performance, and since tile 1 had 7 worker threads instead of tile 0's 5 worker threads, the ratio of data distribution was made to be maxWorkersInTile1/allworkers. As for the situation that tile 1 might not have enough memory to store its part of the image, the system will make sure that data distribution will be as close as possible to the standard ratio. After thorough consideration we found out that the main bottleneck of the system was the maximum speed that the distributor was able to serve workers with. Using one distributor serving all the workers lead to some workers wasting time waiting for their turn. Thus we decided going with two distributors and instead of having a single distributor communicating with every worker in both tiles, the only communication over tiles is between the two distributors. For the two distributors to be able to process the image, each one requires to know about the last and top rows of the other distributor, known as ghost rows This communication with lines takes place with the completion of each round. Both distributors then, begin sending lines to their worker threads along with the required ghost rows. Once a worker has finished processing its chunk of the image, it sends back the altered line and gets more data, if there is any available. To make sure that a line of the image is not modified before the next worker gets the chance to receive that line as its ghost row, the last row that is allocated to a worker, is also copied to a small secondary backup array. At any moment, that array holds the line that will be used as a top ghost row by the next worker to request data.

Workers: To speedup computation and reduce the cycles needlessly spent on empty parts of the image, we implemented an algorithm that first will go over a line and mark which parts need to be computed(dynamic computation). This is decided based on the current cell being checked, the cell above that, and the cell below. If only one of those three cells is alive, then the cell on the current index is marked for computation. If two or three of the cells are marked, then the cell before and the one after it are also marked for further computation. This is not guaranteed to reduce the time spent on blank parts to its minimum but as shown in the tests section below, it significantly speeds up the system. Another talking point for increased performance of workers, is keeping track of cells that will be needed for the next computation, meaning that data already read for a specific computation but will be needed for the next as well, will be passed to the next process to avoid reading them again and again. This way we reduced the computation time by a constant factor.

Output and pausing: Output and pausing are implemented in a similar manner. Once the distributor gets a request from the accelerometer to pause, it sets pauseRequested flag to true. Once a round is completed, this will trigger a pause. Likewise, when Data out requests for an output after a button press event, an outputRequested flag will be set and trigger an output once the round is completed.

RLE: One of the most important aspect of our implementation is the feature of run-length encoding (RLE). This allowed for a significant speed up in reading the image and it optimised the time needed for it. The image is decompressed while being read and stored in an array of unsigned characters in the exact same way as a normal PGM file would have been stored.

Rotation: Finally, one of the features we added to this implementation, is rotation of the image when needed. This is as a safety precaution to keep all of the workers working without any of them slacking off. In order for the system to decide whether to rotate the image or not, it checks the height. If the height is too small, such that not all of the workers correspond to a line, and the width is greater than the height then the system decides it needs to rotate to make use of as much of the workers available as possible. Needless to say, the image is being rotated back to its original orientation before being outputted. When an image is calculated to have a larger width than a line buffer can hold, this will also trigger the system to rotate the image. This has the disadvantage of significantly slowing down I/O operations due to the fact that the image has to be read vertically from top to bottom rather than left to right, and therefore the buffer has to be discarded in every single step.

Tests and Experiments:

Computation of the whole image versus dynamic computation: As it can be seen in the table, the speedup occurring by selecting to compute only non-blank parts of the image is more significant when images are large. The reason for that is that large images and most of the Game of Life patterns, will stabilize and lead to alive cells being more sparsely distributed. Images that exhibit infinite growth, such as a glider gun, will slow the system down, as more and more cells are generated but speed up again, when the cells generated circle back and disrupt the generator. Details shown in Table 1.

Asynchronous channels versus interfaces: Initially, the system was implemented using asynchronous channels, instead of interfaces. The need of interfaces was shown when the available cores to be utilised as worker threads were decreasing. With the aid of interfaces and the distributable feature, we were able to save some cores by making use of a single core to perform multiple tasks rather than allocating a whole core to a task that does not need to be active throughout the whole procedure. This way we gained the availability of three extra cores, thus the system is, without a doubt faster rather than a system using channels. Details shown in Table 2.

Read with RLE encoding versus without RLE encoding: With a compressed image, the system has the advantage of getting a head start. This is due to the fact that reading a compressed image the system is able to get the necessary information to reconstruct the same image faster than a decompressed large image. Even though, doing this with a small image will not lead to a significant difference in time taken, this will always speed up the operation whatsoever. Details in Table 3.

Critical Analysis:

Using dynamic computation, the program is really fast with a staggering range of 7000000 up to 15000000 cells per second, in the best case scenario. Without dynamic computation the program falls to 2000000 cells per second which is less than a third of the previous calculation.

Ways to improve our current implementation:

- Find a way for the image to be compressed throughout the whole process
- Make the workers more intelligent by giving them presets on how specific patterns evolve before hand.
- Make the workers understand when a pattern is ‘still life’ so that they do not waste any process time on that part of the image.

Andreas Georgiou ag14774
Constantinos Theophilou ct14872

Our program can take several sized images ranging from 1x1 up to 1826x1864 pixels. In fact this system can take any image of size (height+4)*ceil(width/8)<=426390 bytes and can still process it. Though if an image with width ceil(width/8)>233 bytes is to be processed, the system will automatically rotate it as mentioned above in the rotation paragraph.
As for how fast can the system evolve the Game of Life it depends on many variables, for example, the size of the image, whether the lines can be divided fairly to all the workers or the percentage of unused empty cells.

Table 1

	Dynamic Computation (cells/sec)	Whole Image (cells/sec)
1x1	140481	107037
16x16	4390945	2225816
64x64	7911103	3128677
128x128	8806332	3404438
180x250	10729910	3469409
256x256	9527824	3411584
512x512	9338283	3488929
1826x1864	12009368	3476895

Table 2

	Interfaces (cells/sec)	Asynchronous Channels (cells/sec)
16x16	4390945	3187464
64x64	7911103	5498456
128x128	8806332	4511475
250x180	10729910	7498416
256x256	9527824	4284629
512x512	9338283	5008959

Table 3

	PGM	RLE
16x16	305ms	203ms
64x64	3393ms	972ms
128x128	13700ms	6797ms
250x180	36424ms	304ms
256x256	53400ms	27573ms
512x512	211700ms	14419ms

Sample images shown below: