Daniel Davies dd16785@bristol.ac.uk MEng Computer Science COMS20001_2017
Chetan Mistry cm16161@bristol.ac.uk MEng Computer Science COMS20001_2017

# Game of Life- Daniel Davies & Chetan Mistry

## Functionality & Design

*Method:* The implementation solves the Game of Life problem using a single farmer thread, which acts as the commander of a number of worker threads. The number of worker threads has been made variable with a #define statement at the top of the program, which makes use of a generic framework that divides the board up equally between the number of workers. The board itself is split up into rows: for example, an image size 64x64 and 4 workers means each worker receives 16 rows to process. The assumption supporting this is that, since each worker runs the same process, they will take roughly the same time to process equal parts of the board.

*Image storage and processing:* Immediately on reading in of the images they are packed into a bit representation, and bit shifting functions are used to extract and write information to the ints that store the data. We have specifically used ints as the assumption is that these are best for the processing speed, since ints are the optimised data type for the CPU, as they correspond directly to the size of the registers. Storing the bits in ints allows us to process images of up to 1024x1024. Tests are provided overleaf. Note: if image size is 16x16 ints are used, but with 16 trailing 0s at the end.

*Hardware:* The hardware has been implemented exactly as required by the specification, using the following methods (which each also handle the correct LED behaviour):

- **Buttons**: A separate button thread is held in the program as a commander thread for when file I/O is to take place. Initially the thread blocks the program until SW1 is pressed. After this, it indefinitely loops waiting for SW2 to be clicked, in which case it commands the distributor to unpack the bit representation and dispatch to the file. (Note**:** for convenience, we output the result of the computation on the very last round in any case).
- **Board tilt:** Again, this is handled by a separate commander thread that continually reads the accelerometer. If the board has been accelerated to beyond -50m/s^2, then the commander halts the workers via communication until the board is placed back down, at which point workers are restarted.

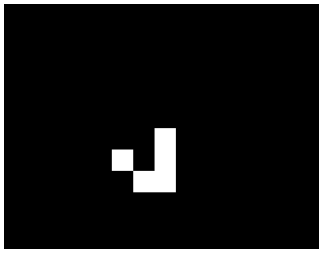
*Timing:* XC timers have been used to test the program. Since timers on the tiles are unsynchronised, one timer thread holds the timer and dispatches it to the rest of the program that requires it. The time is held in a float to handle overflow, since an unsigned 32 bit in can only hold ~42 seconds, and unsigned long long is unsupported in the hardware. Floats however give us the ability of ~10-^38, which is ~10^30 seconds, which is more than enough. However, the timer itself can overflow, as it is held in an unsigned int, hence we check for overflow by doing (previousTime == currentTime) and subsequently adjusting our float appropriately.

*Data exchange strategies:* As mentioned above, we are able to process images of up to 1024x1024. Multiple strategies were used to get to this point, as discussed below:

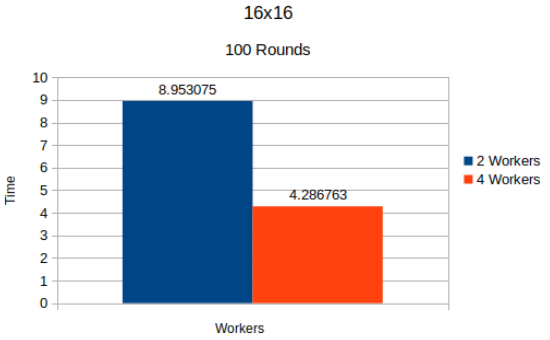
- Initially the workers held their own, full copy of their part of the world. However, this meant that, even with bit packing, there was not enough memory for large images.
- Hence the next strategy was to only store three rows in the workers: the row to be processed and an accompanying top and bottom row, aka, the very minimum required to process a single row. We then refreshed these every time. We did this in two different ways: an optimised and unoptimised method, both described in the testing section, which also explains how these exchange strategies affected performance.

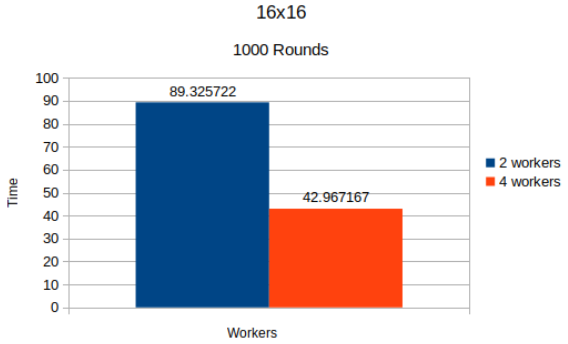
Daniel Davies dd16785@bristol.ac.uk MEng Computer Science COMS20001_2017
Chetan Mistry cm16161@bristol.ac.uk MEng Computer Science COMS20001_2017

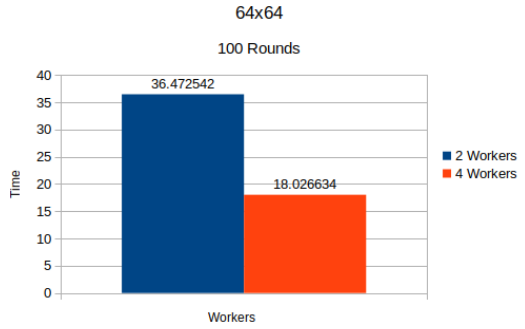
## Tests and Experiments

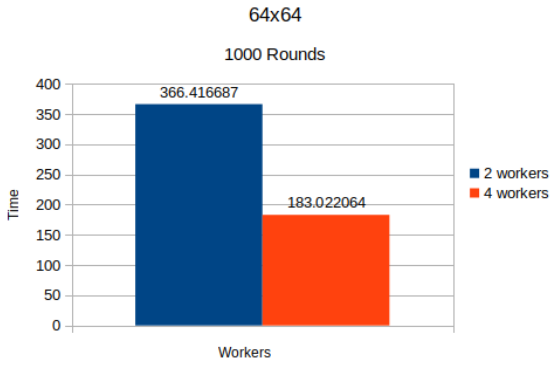


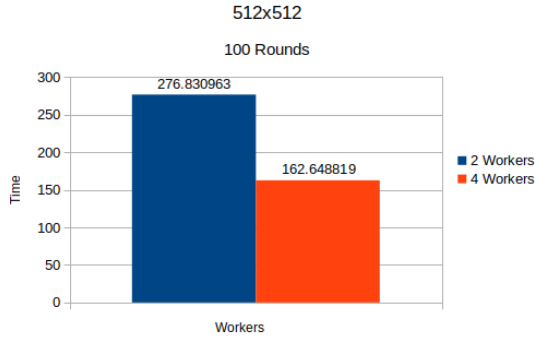
The picture to the right shows the result of processing the 16x16 image after two rounds, below shows all of the test images (as well as a 32x32 image we created) processed after two rounds. We also processed all of the images for 100 rounds and 1000 rounds and compared the time taken to process each (excluding input and output) when using 2 workers and 4 workers concurrently. The difference in time can be shown in the graphs below.

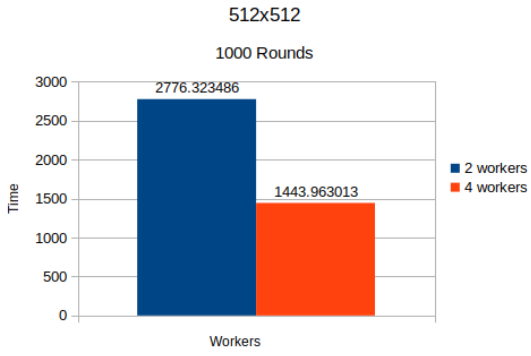



By looking at the graphs we can see that for all image sizes, using 4 workers was almost twice as fast as using 2 workers for 100 and 1000 rounds, this means that our system is incredibly parallel and the more workers we have, the faster we can process the image.

One limitation of the implementation of our system is that we can only have at most 4 workers because when we tried to use 8, we ran out of chanends, cores and timers.

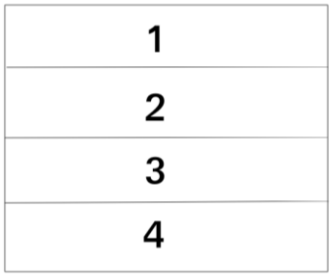
One other limitation of our implementation is that we have to piece everything together sequentially, this is due to the fact that in XC, data structures cannot be manipulated by multiple threads at the same time, and all elements of an array are encompassed in the data structure that is the array. This means that (without using unsafe pointers), we cannot improve the speed at which we take the output data from each worker and use it to update the output image.

Daniel Davies dd16785@bristol.ac.uk MEng Computer Science COMS20001_2017
Chetan Mistry cm16161@bristol.ac.uk MEng Computer Science COMS20001_2017

Another problem with our system we found whilst testing is that memory is used up in interesting way. For 1 worker (i.e. a sequential program) for large images we would run out of memory since everything would end up on 1 tile. So by extension, by having more workers reduces the memory usage on any 1 tile since it gets split up. However, due to some variable declarations, we noticed that having 8 workers would use up more memory than 4 workers for the same image size.

An important factor which contributes to the fact that using 4 workers is almost twice as fast as 2 workers for all image sizes and processing rounds is that our workers have no communication at all since the distributor gives all the workers just what they need to process a cell and so don't rely on one another and are entirely independent, this makes our system embarrassingly parallel.

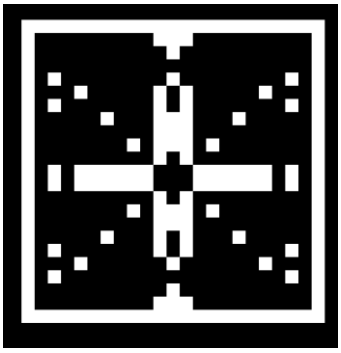
*Data Exchange Strategies tests:* The unoptimized system involved sending multiple lines repeatedly to workers, if we use the diagram below as an example; if we wanted to process line 2, we would send rows 1, 2 and 3 to the worker. Then after, if we wanted to process line 3, we would send lines 2, 3and 4 to the worker which is a waste since the worker already has stored, lines 2 and 3.



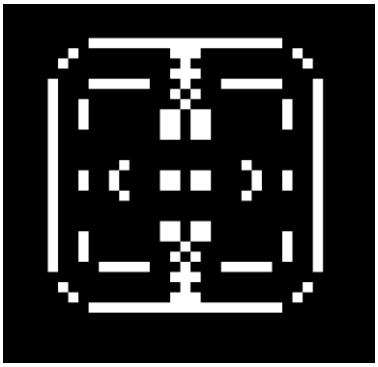

So instead, at the end an indexing system was used. In this method, we keep a pointer to each of top, mid and bottom rows, where mid is the row being processed. After processing, the next row from the worker is loaded in to the "top" row, since this row is now obsolete, so we replace it with the new one. The indices are then shifted: aka the mid now points to what used to be bottom, top points to what used to be mid, and bottom takes on the new row which has been loaded in recently. Now one row at a time only is loaded in, with no duplication.

The optimised version took almost exactly the same time as the unoptimised version, even with the duplicated communication. We think this may be for one of two reasons: only 2 extra communications happen for every line. Communication happens in minute fractions of seconds, so it may not make a big enough difference for the timer the number of workers almost exactly halves the amount to pick up. The other alternative is that, under the hood of the program, since we send different parts of the world, the communications between a worker and distributor could themselves be happening in parallel, rather than sequentially as we thought, taking the same time.

32x32 Image Created

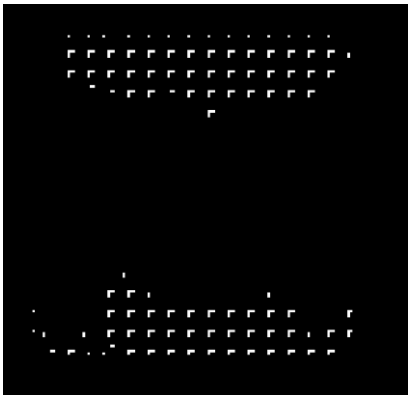


32x32 Image after 2 Rounds

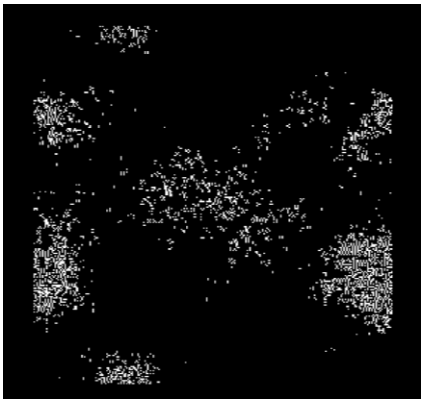


64x64 Image after 2 Rounds

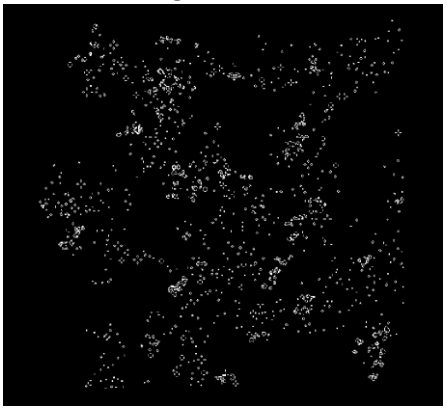
128x128 Image after 2 Rounds



256x256 Image after 2 Rounds



512x512 Image after 2 Rounds



## Critical Analysis

*Performance:* Overall, we found that the actual processing time, even for the larger images of 512 and 1024, are fairly fast, as shown in the testing section. We found that our implementation is also very scalable: the workers are designed in a completely embarrassingly parallel fashion, shown by the fact that doubling the number of workers halves the processing time, meaning that if we had larger images to process along with more powerful hardware, the program would be able to handle progressively larger images fairly well by adjusting the number of workers.

Unfortunately, however, the xmos board is limited by both cores and channels and so only a fairly small number of workers could be implemented in the end. Our system could only go between 1 and 4 workers before the cores and the chanends were exhausted. Attempts were made to get 8 workers, but while we managed to keep the number of chaneneds below the limiting value, we still ran out of cores when trying to implement the 8 workers. If we had more time, we would take a look at reducing the number of cores used by the program to see if we could save enough for the program to run. One method of doing this would be to make some of the functions [[combinable]] and place them on the same core, allowing for sharing of resources through scheduling. This could work for some functions as they end in a while(1){select{…}} structure and return void. Another possibility we believe however would be a smarter distribute function: currently we use a par for block to distribute work to the workers. The issue with this however we think is that each process in the par for is allocated a thread to run on a core: this means that for large numbers of workers, a large number of threads would possibly be allocated to separate cores, thereby removing resources. If this was done in a normal for loop however, only one thread would run and distribute the work to the workers, meaning we would have more cores available and could potentially run 8 workers. Furthermore, as the number of workers scales the returns of using a par block diminish as the cores have to implement process scheduling, and therefore we wouldn't expect too big a performance difference.

*Future improvements*

*Bit packing:* One large assumption in the program is that ints are assumed to be the most efficient for processing in the CPU and hence give the fastest results. However, this may very well not be the case: other data types, such as uchar for example, may very well end up more efficient to pack the bits, and would also provide a convenient method to pack bits that are multiples of 8 but not 32, for example, the 16x16 image could be packed into rows of 2 uchars rather than one int, avoiding the need for 16 trailing 0s. This is definitely an aspect we would like to test if we had more time.

*Workers-timers:* Currently we have a separate channel from the timer to each of the workers. This was simply for accuracy: each of the workers receives an accurate poll from the timer directly, and is easily implemented with channels. However, this scales very poorly: for every worker, we have an extra channel, meaning for 4 workers, there are 8 chanends used. We could have avoided the creation of all of these channels however: the distributor also has a channel to the timing thread, and therefore to save the channels to the workers, a system could have been implemented where firstly the distributor polls the timing thread for the time, then routes it to the worker when it is required by the pause function. This is easily implementable with select case statements and would follow examples of what we have previously already done. The only issue would be the data type: the distributor-worker channel is in fact an int channel, but time is a float, hence to distribute the time to the worker, two or more ints would be needed to be sent in replacement of a float. (In fact, this kind of system was already implemented in an older version of our code, but didn't make it to the final version because of how much easier direct channels are).