Alex Robinson (ar15247@bristol.ac.uk), Luke Storry (ls14172@bristol.ac.uk)

Cellular Automaton Farm Report

Functionality & Design

Overview

We have implemented a concurrent solution which uses one tile for control and the other for logic. Our control code consists of processes for checking for orientation change, button presses, reading images, and writing images. We contained all the logic to a single tile in order to utilise faster memory access of pointers and shared memory, as opposed to sending data across interfaces every tick. Once an image is read, a Farmer process splits the image into horizontal strips, padding each with overlaps. The Farmer then assigns each Worker a strip and monitors its work, keeping Workers in sync each tick. Each Worker loops through each cell in their strip, calculating its new state, before writing back to the shared memory.

Due memory limitations on the xCORE-200 eXplorer board we decided to use bits to represent the state of each cell. Our Image Reader reads 32 cells at a time, and packs these into a single integer, which is then used throughout and later unpacked on write. The advantage of this is twofold, it reduces memory required to store the entire board allowing larger images to be processed, and secondly it reduces communication overheads as more information is stored in less data.

Another feature of our solution is that instead of each Worker and Farmer having an entire copy of the image, only one full copy exists, with each strip shared with a Worker. Each Worker then only requires a small copy of its strip for reference as it calculates and writes directly back to the original memory. The ownership of these shared memory blocks is transferred between the Farmer and Worker at well-defined locations in the code to prevent deadlock. Ideally Movable Pointers would have been used here, but they aren't fully compatible with 2D arrays.

Farmer each tick:

- Check for start read/write (from button) or pause/play (from accelerometer) signal
- Update strip overlaps and tell Workers to start, giving them access to the shared memory
- Wait for Workers to complete
- Update timers and tick count
- Repeat

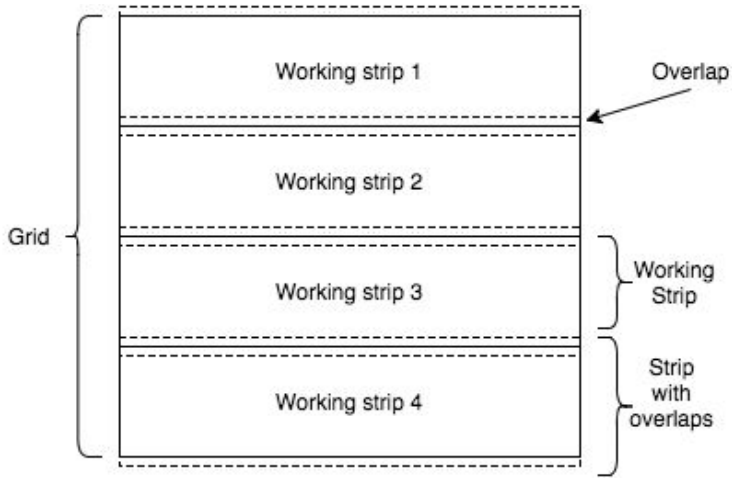
Worker each tick:

- Wait to start work
- Copy strip into a local reference copy (required to calculate and immediately write back new values)
- Loop through strip (skipping top and bottom overlap rows) calculating new cell value and keeping track of the number of live cells for stats
- Tell Farmer that work has finished.

Work allocation

Our system has a fixed number of Workers, because memory allocation has to be done as soon as the program is run, before any images have been read. Each Worker is allocated a strip which it is responsible for calculating the new values for. A strip represents a collection of rows of cells, the number of rows (height of the strip) is determined when an image is read in. The system attempts minimise the size of each strip in order to maximise the number of Workers it can use (and therefore the amount of work it can do in parallel). A strip contains an overlap from the strip below and the strip above, the overlap contains a copy of the bottom/top row from the above/below strip. This is required as in order to calculate the next value of a cell at the top of a stirp it needs to know the row above. Each tick after all Workers have completed their strip the overlap rows are updated with memcpy.

Strips were chosen as they minimise the amount of communication required between Workers a only the overlaps need to be updated each tick.



Outline in CSP

```
FARMER       = CHECKBUTTONS; (ReadDone → WORK; FARMER) □ (FARMER)
CHECKBUTTONS = (SW1 → ReadImg → SKIP) □ (SW2 → ExportImg → SKIP) □
                 (Orientation → Pause → SKIP)
WORK         = WORKER ||| WORKER ||| WORKER ||| WORKER
WORKER       = CalculateStrip -> SKIP
```

Tests & Experiments

Initially we designed our system on paper, and then translated to a quasi-parallel solution Python in order to prototype quickly. Prototyping with a testing suite in Python allowed us to rapidly find bugs with our initial logic before we translated into xC. For example, not storing a reference to the old array and updating the grid in place which caused the game of life to calculate incorrectly. Also a lot of issues with calculating overlaps and using modulo to cope with the wrapping indexes were solved quickly in Python

Originally our system stored each cell as an integer, but this proved to reduce performance considerably as it had high communication overheads. We re-designed, and implemented a solution to instead use only a single bit per cell. On read we pack cells into an integer (32 bits on the board), then store an array of integers to represent each strip, making the board a 2d array of strips. On write they are unpacked to write to the file. Not only did this improve performance but it allowed us to process even larger images.

We also experimented with movable pointers and using Interfaces for memory exchange between the Farmer and Workers. We found that using pointers to shared memory considerably boosted performance. However due to limitations with xC we couldn't use movable pointers with 2-d arrays and so instead directly wrote to memory being careful to transfer ownership at well-defined locations in the code.

We found read times of images were slowing down development/testing so we experimented with different code optimisations to increase performance. One optimisation was to move reading process onto the same core as the Farmer (rather than using a separate process and communicating via interfaces), we hoped this would reduce communication overheads however our results showed no significant change. We also tried optimizing the reading code to remove two loops that converts characters into an array of ints and then into bitfields, however we found that the compiler had already optimised this as the change made no significant difference.

Result of the given 16x16 image after 2 rounds: See appendix *16x16 2 tick*

Size	Read Time / s	Time 100 Ticks / s	Live cells	Result
16x16	0.520	0.300	5	Appendix *16x16 100 ticks*
64x64	4.022	4.200	463	Appendix *64x64 100 ticks*
128x128	14.981	17.247	1535	Appendix *128x128 100 ticks*
256x256	59.551	68.505	4659	Appendix *256x256 100 ticks*
512x512	243.102	275.597	6739	Appendix *512x512 100 ticks*
567x700	N/A (Generated on board)	255.814	5	Appendix *567x700 100 ticks*

Critical Analysis:

Virtues and limitations

As each Worker is given the same amount of work to do per tick, processing time is constant and therefore it is possible to predict how long an image will take to evolve some number of iterations. This can be seen in the results table where *Time 100 Ticks / s* almost quadruples as the image size doubles. See *System Performance* for more details.

Our system allows repetitively reading and writing, and continuing to processes the board. On read the system will recalculate strip sizes and reallocate to Workers, allowing images of multiple sizes to be input and processed in the same run. Furthermore the system dynamically allows non-square images, as evidenced by the final image in results table.

The major downside of the Shared-Memory implementation is that we cannot spread Workers out over two cores, so are limited to just using seven Workers on a tile with the Farmer. This limits the number of Workers, but theoretically speeds up the communication.

System Performance

The max size of a processable image with our program is 820x820, due to memory constraints. Our model pre-allocates 3000 ints per Worker (this number was reached via experimentation), our maximum number of Workers is 7, and we can store 32 cells in each int (as we use individual bits to store cell state). So in total we have $3000 * 32 * 7 = 672,000$ cells so roughly an image of size 820x820.

Tick speed is constant and proportional to the image size, as each Worker is given an equal sized strip and calculations on the strip take a constant time. To approximate the amount of time per tick for a grid of any size you can use $0.0000103125 * \textit{number of cells}$, as determined by test data above.

Improvements

Due to time constraints, we were unable to fully realise our performance visions. However we did design a number of methods to increase performance. One bottleneck we have identified is in our memory management for Workers. On each tick each Worker copies its strip into a local copy, the reason for this is because an unmodified copy of the strip is required to calculate the next value of a cell. This copy incurs a large time penalty, in order to reduce this, a system could in which on read a Worker is given its strip, on each tick the Worker would update a blank strip with the newly calculated values (referencing the old strip), on completion the Worker would swap the pointer points to the old strip and the strip its writing into. This avoids a copy being generated on each tick and would increase performance dramatically.

We designed a multi-tile memory solution, but did not have time to implement it. This design included a Farmer core on each tile, which would manage its Workers on the same core. The grid would be split in half with strips across both tiles. Each Farmer would perform overlap swaps for Workers on their tile, for the two overlaps which go across tiles these would be passed via interfaces between the Farmers on each tile. When initially reading an image, the system would prioritise just using strips on a single tile rather than across two in order to reduce the inter-tile communication overhead.

Appendix