COMS20001 - Concurrent Computing - Coursework 1

map ("Adam " ++) ["Beddoe", "Fox"] — ab16301/af16371

1 Functionality & Design

The xCore-200 eXplorerKIT implementation of John Horton Conway's Game-of-Life implements the following functionality:

- Correctly evolving Game-of-Life, according to the given rules
- 2, 4 or 8 worker threads
- Button, board orientation and LED behaviour
- Ability to process larger than 512x512 images with memory on both tiles
- Timers to measure processing speed and comprehensive unit testing
- Ability to generate & process random images up to a size of 1472x1472 on the board
- Dynamically chooses implementation based on input size

1.1 Method

In order to iterate the game of life, the program works as follows: The main function starts up all threads, specifies which tiles they are on, and assigns channels between them. The distributor function is responsible for the mechanics such as LED behavoiur, accelerometer pausing and telling the workers to input and output the current game state. To minimise thread communication, the workers pass board state directly to the IO threads when required and only communicate with the distributor to send a confirmation signal each round and to send the count of live cells when the game is paused. The workers remain in sync as they require synchronous communication to share columns each round.

1.2 Worker Types

Each of the worker threads receives an equal column of the total image. There are two different worker systems, the program dynamically chooses the best implementation based on the size of the input image:

1.2.1 Unpacked Worker

This worker reads in the segment of the image it is passed from the IO stream and stores it in a uchar array, where each element can be 1 or 0 and represents an alive or dead cell respectively, with an extra column on each side to store the state of left and right workers outmost columns. The threads communicate in pairs, passing or recieving first depending on whether they have an odd or even ID. They then proceed to make a copy of the array, and then iterate on the original based on the sum of the neighbours of each square, the added columns in the array from each worker meaning that the solution requires no further communication until the next round. It will then continue iterating and communicating with the adjacent threads until told to perform one of the IO actions by the distributor.

1.2.2 Packed Chunk Worker

This implementation uses bit packing, and splitting the image up into chunks, to make it optimal for reducing memory usage, allowing for large images. Packing into an unsigned char (uchar) we can store 8 values in each, drastically reducing the memory required but adding computational complexity by computing the packing and unpacking, and the copying of edges between chunks each round. The *left, right, top, bottom* variables, hold the corresponding row of the neighbouring chunk and the 4 least significant bits in *corners* to store the 4 adjacent corners. The storing of the neighbours for each chunk rather than the whole array means that the base array takes up more space, however only

one chunk needs to be loaded into memory when iterating, rather than making a copy of the entire worker's array.

2 Tests & Experiments

2.1 2 Round Images

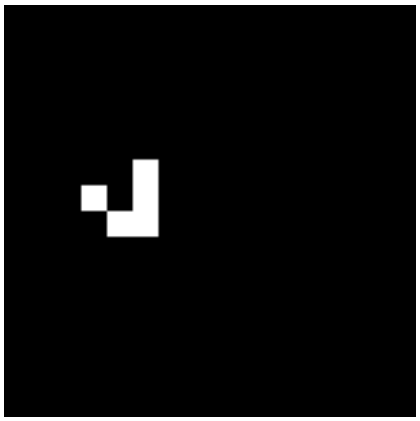

Figure 1: 16x16

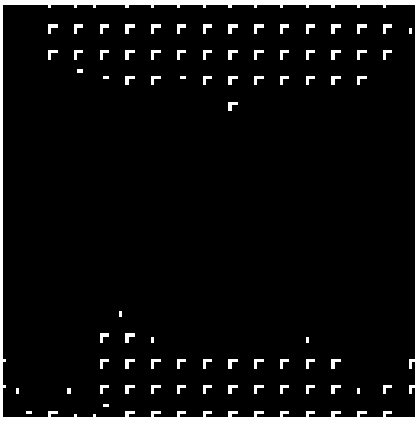

Figure 3: 128x128

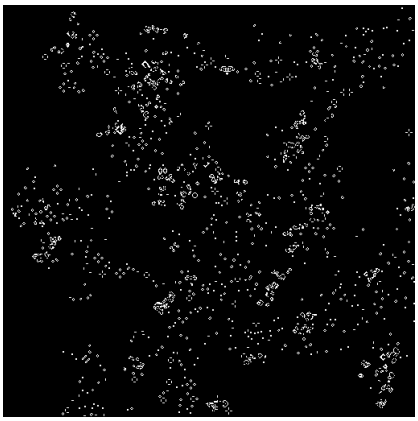

Figure 5: 512x512

Figure 2: 64x64

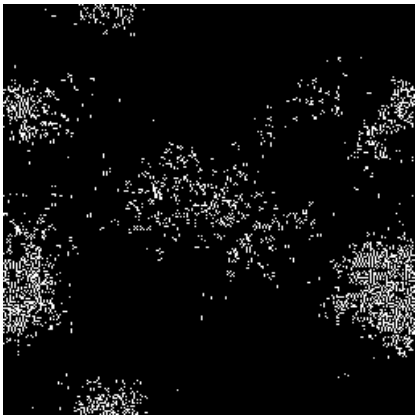

Figure 4: 256x256

3 Critical Analysis

Through testing, it was determined that the additional time complexity of packing and unpacking meant that the