# 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 mechanics such as LED behaviour, accelerometer pausing and instructing 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. An extra column on each side stores the state of the left and right worker's outmost columns. The threads communicate in pairs, passing or receiving first depending on whether they have an odd or even ID. They then proceed to make a copy of the array, and iterate on the original, based on the sum of the neighbours of each square. The additional columns mean 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, making it optimal for reduced memory usage. This allows for large images, in this case up to 1472x1472. Packing into an unsigned char (uchar) it is possible to 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 edge of the neighbouring chunk and the 4 most significant bits in *corners* 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 Packed vs. Unpacked

Across the same size images, as shown in *Table 1 & 2*, the unpacked implementation is faster than the packed one. By packing communications before they are sent on a channel, communication time is reduced. However packing the data incurs additional computation cost. Furthermore, the implementation requires copying of the edges between chunks for each iteration. A combination of these two effects explains the results in the tables.

It was also expected that, as image size increased, the packing approach would eventually outperform unpacked, due to increased communication. As seen in *Figure 2* this is not the case; the computational packing cost always overshadows the packed communication performance improvement.

## 2.2 No. of threads

Looking at *Figure 1* it can be seen that the performance gains from a greater number of threads is more evident for the packed worker. As the number of threads increases, the overriding limiting factor is the communication between worker threads. The packed worker achieves greater gains as less data is sent over the channels. Though it is not possible to test with the current implementation, it is suspected that increasing the number of threads would eventually decrease performance, for the reason previously mentioned.

The program does not allow more than 2 threads to operate on the 16x16 image, however this is not problematic for such a small image as more data would be communicated between threads than is operated on within them. This would decrease performance.

## 2.3 Inter-tile vs. Intra-tile communication

When performing benchmarks on inter/intra tile communication, it was discovered that, while intra-tile is faster, the effect is almost negligible compared to other limited factors. When tested on the provided 128x128 image utilising 2 threads, intra-tile was only 0.146% faster.

## 2.4 Tables

| | | No. of threads | | |
|---|---|---|---|---|
| | | 2 | 4 | 8 |
| Image size | 16 | 0.029248 | N/A | N/A |
| | 64 | 0.461743 | 0.233838 | 0.190104 |
| | 128 | 1.839617 | 0.918425 | 0.74095 |
| | 256 | 7.08548 | 3.548166 | 2.81206 |

Table 1: Unpacked Worker

| | | No. of threads | | |
|---|---|---|---|---|
| | | 2 | 4 | 8 |
| Image size | 16 | 0.036808 | N/A | N/A |
| | 64 | 0.61281 | 0.30677 | 0.227792 |
| | 128 | 2.447382 | 1.223531 | 0.972146 |
| | 256 | 9.756354 | 4.853463 | 3.880383 |
| | 512 | 39.461591 | 19.442143 | 15.49954 |
| | 1024 | 152.395147 | 77.946608 | 62.25204 |

Table 2: Packed Worker

## 2.5 Graphs

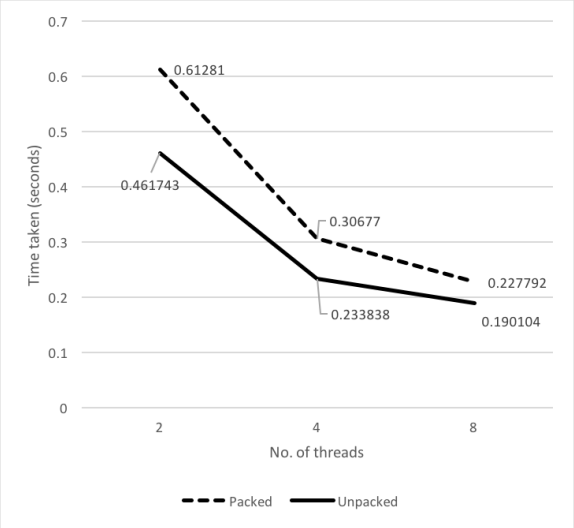


Figure 1: 64x64 - Packed vs. Unpacked

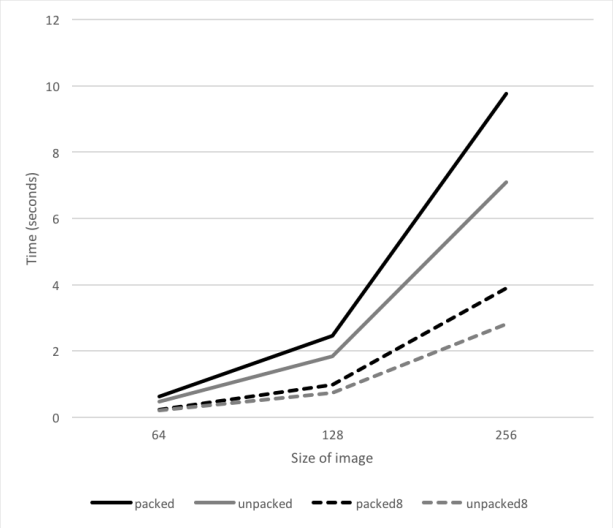


Figure 2: 256x256 - Intra vs Inter

## 2.6 2 Round Images

As required, below are the second round images for all of the provided example images. Additionally, *Figure 6* is an example of the largest possible image (1472x1472).

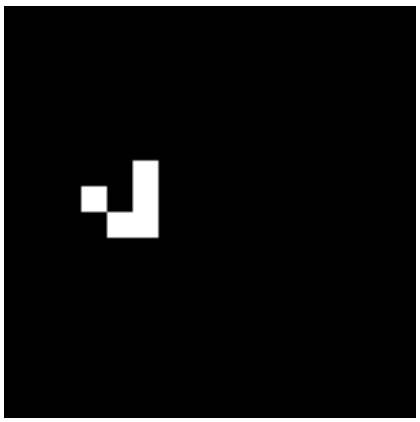

Figure 3: 16x16

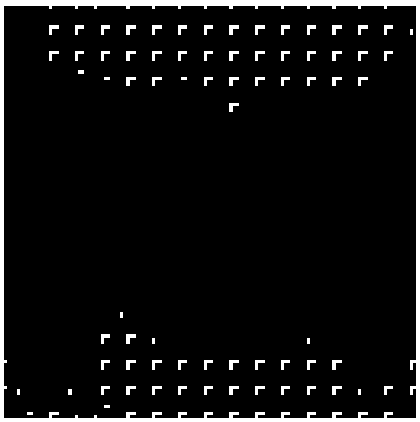

Figure 5: 128x128

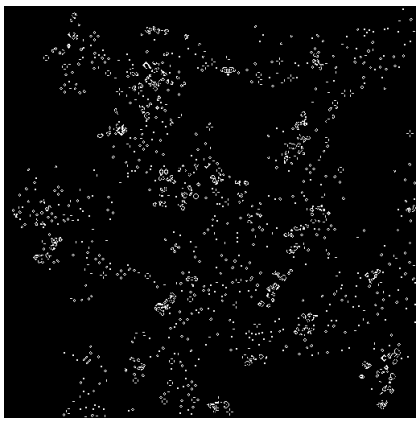

Figure 7: 512x512

Figure 4: 64x64

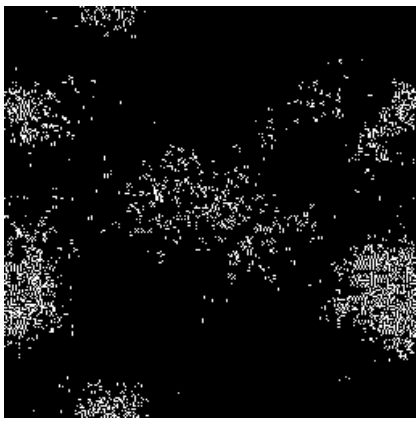

Figure 6: 256x256

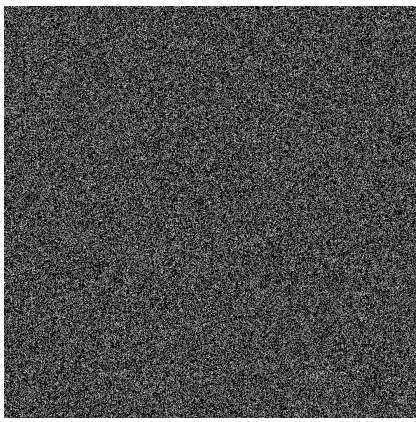

Figure 8: 1472x1472

# 3 Critical Analysis

With the -O3 compiler flag and correctly distributed functions across two tiles, the program can achieve a maximum image size of 1472 x 1472. The fastest it can iterate a 16x16 image is with two threads at approximately 2700 rounds per second.

It was initially expected that the packedChunkWorker would be the fastest implementation for all but the smallest images. Through benchmarking, it was determined that the additional time complexity of packing and unpacking, and inter-chunk edge sharing, meant that the packedChunkWorker was slower than the unpackedWorker. Implementing the dynamic selection of the worker function based on file size means that the program uses the fastest available and will use the unpacked version up to file sizes less than or equal to 256 x 256.

Another dynamic improvement that could be made, which XC does not currently allow (but should), is the ability to decide which tiles the functions will run on dynamically. Currently, the program will run half of the worker threads on tile 0 and half on tile 1, which means two threads will have to communicate cross tile. This is required by our implementation in order to make use of the memory on both tiles for larger images, however, it is slower for images where this is not required. Even though cross-tile communication is only marginally slower than intra-tile, it would still be advantageous to place all the threads on the same tile, in the case that the image is less than or equal to a 256 square; which is known at compile time.

The memory usage of packedChunkWorker, and thus the maximum image size, could be greatly improved by packing into ints or longs, rather than uchar. This is because it currently requires 5 extra bytes for every 64 grid bits stored, but by packing into an int, it would require only 5 per 1024 while still not needing to copy the entire grid for iteration.

Another improvement that could be made would be the creation of a third dynamic worker, which iterates in smaller packed columns rather than chunks. This would not be as memory efficient, but would have linear complexity on size of image for sharing between sections, as opposed to squared complexity for chunks.

It may have been faster, for some medium sized images, to only pack the communication between threads and store unpacked values. This would greatly reduce the channel communication overhead between sharing threads at the start of an iteration, but still remain fast for the summation calculations. However as it has not been implemented or benchmarked, it is unclear whether the packing overhead would have negated any performance benefits for images of that size.

The program is limited in how the number of threads must be a power of 2. This means that it was not possible to test 10 threads. The input size is also quite limiting, as it must be a multiple of the number of threads (a multiple of the number of threads * 8 for larger than 256). However, implementing these changes would greatly increase the complexity of the code, and potentially decrease efficiency.