Igor Dolecki

2G400 BSc Computer Science

id17002@bristol.ac.uk

Todor Hristov

2G400 BSc Computer Science

th17418@my.bristol.ac.uk

Game of Life Report
Team 17

Introduction

For this assignment we have written a concurrent implementation of John Conway's Game of Life. It is written on xC and runs on xCore-200. It makes use of the message passing model and employs 8 worker threads for the computation of the next state of the board. It makes use of bit packing and the memory of both tiles in order to handle gamestates of up to 1736x1736 cells.

Functionality and Design

Timing

The timing is done by two threads. The mainTimerThread communicates with the distributor. The second timing thread (helperTimerThread) checks for overflow every 100 milliseconds and notifies the main timer thread of any overflows. This lets us measure the time with the precision provided by the hardware for periods longer than a minute.

File Input and Output

Reading and writing from files is done through two threads: dataInStream and dataOutStream. The two threads are not supposed to be used concurrently and their performance does not affect the speed of computation, which made them suitable to put on the same core in order to make more worker threads.

Button, LEDs and Orientation

The Buttons, LEDs and orientation each has its own thread. When started, the program prints a message that SW1 should be pressed to start reading input file. After pressing it, reading from the file starts, during which the combined green LED is lit on. When reading ends, it is turned off. Then calculation starts, which is indicated by changing the state from on and off of the separate green LED each time calculation of the next state ends. When SW2 is pressed, the current gamestate starts to be exported to a PGM file, which is indicated by lighting on the Blue LED. When writing to file ends, the led is turned off and computation restarts.

The tilting thread sends a signal to the distributor once per each state change. If the absolute value of the tilt is more than 50, then the board is said to have become tilted. If it is less than 10, then it is considered to be leveled. The difference in the two values is introduced to minimize the double detection of state change.

Game State Calculation

The running of the game is done with the help of one distributor thread and 8 worker threads. At the beginning of the calculation the board is split into 8 equal segments (by rows) and each is sent to a worker. The distributor stores locally only the additional rows for each thread in order to save memory. After this on each round only the additional top and bottom rows are exchanged between the distributor and the workers, which minimizes the amount of channel communication necessary. Before exporting to file, the whole table is requested from the worker threads and directly send to dataOutStream.
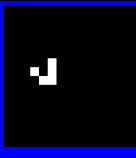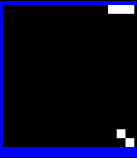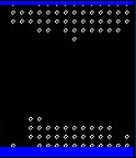
Memory Usage Optimisations

Bit Packing is implemented into "unsigned characters". Additionally the whole game state is only saved in the workers, while the distributor only has the additional rows for every worker. Furthermore, the

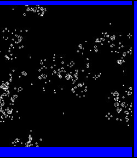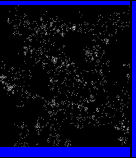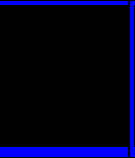workers have only one copy of the the game state, circumventing the need to have an explicit previous and current state. This lets us process matrices of up to 1736x1736 cells.

Tests and Experiments

Tests for the Game State after a Predefined Number of Rounds

	16x16 (2 rounds)	16x16 (100 rounds)	32x32 (100 rounds)	64x64 (100 rounds)	128x128 (100 rounds)	256x256 (100 rounds)	512x512 (100 rounds)	1024x1024 (100 rounds)	1736x1736 (100 rounds)
Time (seconds)	0.000653	0.0326	0.1225	0.4810	1.8863	7.4729	30.1153	120.0322	347.1470
Resulting State									

All tests, with the exception of the one for 32x32, 1024x1024 and 1736x1736 matrices, were performed with the provided test images. The tests for 32x32, 1024x1024 and 1736x1736 matrices were done with the provided 16x16 image state to which dead cells were added.

Test Results after Different Optimisations

	Time per round In seconds	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	1736x1736
#0	Base Test	0.002003	0.007991	0.033530	0.134068	0.521104	MEM	MEM	MEM
#1	No bitpacking	0.000847	0.003238	0.012764	0.050560	0.199992	MEM	MEM	MEM
#2	Bitpacking	0.000625	0.002461	0.010039	0.039354	0.157645	0.627928	MEM	MEM
#3	Two Tiles	0.000608	0.002400	0.009762	0.037695	0.152922	0.606438	2.425435	MEM
#4	Only Additional Rows	0.000595	0.002345	0.009534	0.036808	0.148893	0.590533	2.363247	MEM
#5	Asynchronous	0.000598	0.002342	0.009333	0.036596	0.148334	0.596291	2.388955	MEM

MEM means memory limit error.

Base Speed Tests (#1)
The first row of the table above shows the results for basic one-threaded implementation without bitpacking or any other improvements. It is used as a starting point to show how the different optimisations manage to lower the runtime. The subsequent 5 tests were all performed with 4 worker threads.

Bit Packing (#1 and #2)
This memory optimisation was implemented in order to reduce the size of the memory needed. The tests reaffirmed our suspicion that even though working with bits rather than bytes has an overhead when calculating next state, this is overshadowed by the gain of sending less bytes between the threads.
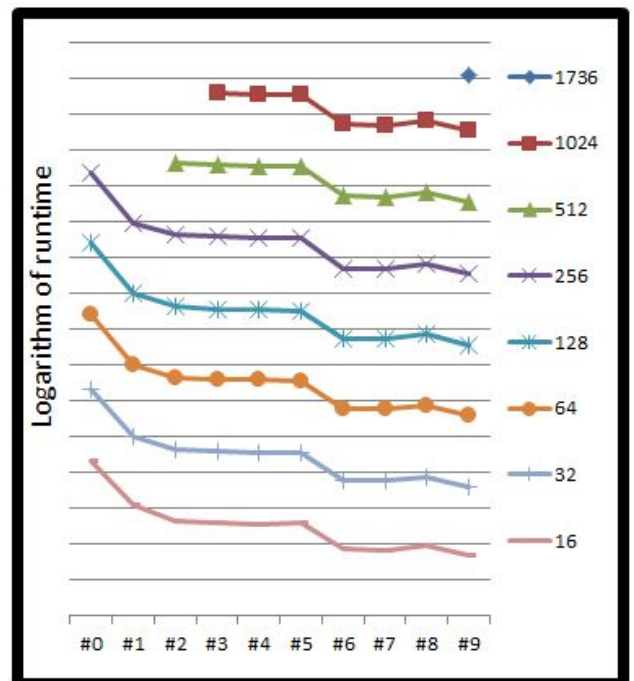
One vs Two Tiles   (#2 and #3)

It was suspected, and subsequently proven wrong, that communication between tiles is a lot slower than inter-tile communications. Test results showed that communication between tiles is a little faster than inside the tile, which was observed persistently over all board sizes. Our interpretation of those results is that channels inside tiles are not completely disjoint from each other, and that their performance is affected by the workload of the other channels on the tile.

Channel Communication Overhead   (#3 and #4)

Tests were performed to check if channel communication overhead is too much of a problem. The amount of information send between the distributor was reduced from the whole table on each round to just the two additional rows on top and bottom. There was a small, but measurable, speed gain, which was smaller than expected.

Synchronous vs Asynchronous Channels   (#4 and #5)

Tests with Synchronous and Asynchronous Channels. Asynchronous channels were found to be slower, and were therefore not used in subsequent implementations.

Different Number of Worker Threads

Tests with different number of worker threads showed that the number of workers is very important for the overall performance. Doubling the number of workers reduced the runtime by approximately 40%:

	Time per round In seconds	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	1736x1736
#4	4 threads	0.000595	0.002345	0.009534	0.036808	0.148893	0.590533	2.363247	MEM
#6	8 threads	0.000367	0.001379	0.005424	0.021377	0.082759	0.337989	1.346576	MEM

Miscellaneous Optimisations

Additional optimisations were implemented and tested. Row #7 provides information when implementing the copying of current state into the previous state matrix with memcpy. Row #8 additionally uses a constant array holding the powers of 2 for faster extraction of the bits from bytes. This proved to be slower than bitwise shift every time.

	Time per round In seconds	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	1736x1736
#6	Normal	0.000367	0.001379	0.005424	0.021377	0.082759	0.337989	1.346576	MEM
#7	Memcpy	0.000358	0.001378	0.005522	0.021327	0.082847	0.328144	1.309630	MEM
#8	+Const Array	0.000392	0.001479	0.005822	0.023164	0.090014	0.358249	1.434841	MEM

Final Solution

The test results for our final solution are shown below:

	Time per round In seconds	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	1736x1736
#9	Final	0.000326	0.001225	0.004811	0.018864	0.074730	0.301157	1.200322	3.471471

Critical Analysis

Computation Time

We believe that our program has room for optimisation with respect to the computation time. Currently, the computation time is not dependant on the amount of live or dead cells on the board, because no optimisations for calculating only parts of the matrix adjacent to live cells are implemented.

We believe that it is possible to exploit the fact that if a cell does not have any alive neighbours, then on the next round we can be sure that it won't be alive. This would enable us to do a Breadth-first search from all live cells on each round and only consider the cells with a distance of 0 or 1. This would greatly improve the performance if the matrix is sparse enough.

Board size restrictions

In its current form, our program only works for images of height divisible by the number of worker threads, and width divisible by 8. This may be improved by dealing with the edge cases. For the purposes of this coursework all of the provided images had dimensions which were a power of 2, so dealing with those edge cases was decided to be circumvented by changing the number of worker threads.

Number of Worker Threads

Our final solution uses 8 worker threads. After testing for different worker sizes we came to the conclusion that the more worker threads we have, the better the performance. We defined the number of worker threads in a define statement in order to let us easily change it to an arbitrary number between 1 and 8, which would let us calculate matrices of height which is not strictly divisible by 8.

Channel Communication

Our program uses synchronised channels for message passing between threads. Even though we initially thought that asynchronous channels would be faster, we got strong empirical evidence that this is not the case. We think that this is caused by the overhead of having a buffer, which overweights the gain that it may provide.

Memory Usage

Our final solution uses more optimizations to lower the memory usage. The game state is only saved in the worker threads, while the distributor only synchronises the additional rows between them.

The worker threads do not hold the whole previous and current state at the same time. They achieve this by explicitly holding the additional top and bottom rows, and an additional buffer of two rows. This buffer progressively moves downwards, temporarily holding the next states of the row before the one for which we are currently calculating, enabling us to safely calculate the next state. Those additional tweaks let us calculate even bigger matrices of sizes up to 1736x1736. A game state of this size, when stored in one bit per cell, takes 376712 bytes. Taking into account that each worker thread would always need additional memory to calculate each subsequent state, and the existence of other local variables, we believe that our program manages to compute matrices of sizes very close to the limit for the provided hardware.