The Game of Life

Callum Ke: in18536
Tom Stark: jb18789

I INTRODUCTION

The goal of this report is to outline the solutions for implementing the Game of Life Cellular Automata using both sequential and various concurrent methods. It also aims to discuss the bottlenecks that occurred and optimisations that were made to improve each stage of implementation.

II FUNCTIONALITY AND DESIGN

1) STAGE 1

SEQUENTIAL: This stage requires a single-threaded solution which iterates through each bit of the board and counts the number of alive neighbours(adjacent cells).

The Game of Life rules are applied to update the new state of each cell in a temporary board. At the end of each turn we swap the board with the temporary board so that the temporary board to ensure the 'aliveness' of the current cell is based around the previous state of the board and not on the new state that is being built.

DIVIDE AND CONQUER: This stages requires implementing a non-memory sharing multi-threaded solution using the message passing model. Buffered input and output channels are assigned to each worker thread. At the beginning of every turn, each worker receives and recreates its own portion of the world, byte by byte, through its corresponding input channel. We also make sure each worker is sent an extra top and bottom row so it can update the edge cells of the worker world.

The new state of each cell is immediately sent to a workers corresponding output channel. The distributor then rebuilds the world using the output channel from each worker at the end of each turn.

2) STAGE 2

This stage requires the implementation of keyboard and timer interactions.

We implement this by using a 'key channel' that sends messages between the control and the distributor. Depending on the key pressed, the distributor will either: pause, output the current world as a new file or terminate. There is also a timed print statement that prints the number of alive cells every 2 seconds that is implemented using a Timer.

3) STAGE 3

As the board size is always a power of two, we can always split the board evenly between threads if the number of threads is a power of two. However, if the latter is not the case, using the same method to send portions of the world to each thread will not work as there will be a remainder of the world that won't be sent(image height modulus number of threads).

To solve this problem, we first test if the number of workers is a power of two. If so we know that the remainder of rows will need to be sent to the *(number of threads)*'th worker and we make sure that this specific worker's world is increased by the remainder of rows.

4) STAGE 4

This stage aims to improve the efficiency of the message passing implementation in Stage 1b using 'HALO EXCHAGE' instead of distributing and rebuilding the world at each turn in the distributor.

We use an array of channels where the i'th index contains an array of two buffered channels corresponding to the top and bottom rows for the i'th thread. The distributor distributes each worker its own portion of the world. Each worker is assigned channels where it sends its halos to and receive its halos from. Every turn, each thread is notified by the distributor to send its updated halos and receive its required halos using message passing. Each thread must **send before receiving** or there will be busy waiting and thus a deadlock.

Each worker only updates its own portion of the world. It sends it back to the distributor to be rebuilt after all turns are processed.

5) STAGE 5

This solution uses a memory sharing multi-threaded approach. This requires each thread to be sent a pointer to the same world/memory address which is then asynchronously updated by each worker.

This may introduce a **data race** where the distributor begins processing the next turn before each thread finishes working on the new state. To prevent this, we use WAITGROUPS to make sure each thread has finished processing the current state before going to the next turn. At the end of each turn after all workers have finished, the distributor swaps the world with the new state of the world in order to be sent to the workers again.

III TESTS, EXPERIMENT AND CRITICAL ANALYSIS

TESTS AND EXPERIMENTS:

1) We first made sure the output images were correct in correspondence to running `make test`. The before and after images of processing 100 turns can be seen in Figure 1.
2) The first experiment we ran was running `make compare` to see the performance of each implementation in comparison to the baseline benchmark solution. The averages for 10 comparisons can be seen for each stage in Tables 1-4.
3) Figures 5 and 6 portray the runtime of each stage in accordance to running `make bench`.
   - Figure 5 specifically shows the comparison of each stages performance with respect to file size over 100 turns and the multi-threaded solutions using 8 threads.
   - Figure 6 shows the comparison between each multi-threaded solution with varying sizes of threads over 100 turns using the 128x128 image.
4) Figure 6 shows the comparison between each multi-threaded solution with varying sizes of threads over 100 turns using the 128x128 image.
5) Using the `make trace` command we noticed optimisations that could be made and bottlenecks that occurred.
   - One optimisation was changing the method of distributing each portion of the world to each thread from the distributor. In the first method, the distributor iterated over all cells and synchronously distributed the cell to the relevant worker. The trace showed that a majority of the time was spent in the distributor distributing cells from PGM.
     – The solution to this was sending each portion asynchronously to each worker using a goroutine per worker.
     – Table 5 shows the performance increase.
   - Another optimisation made was changing the method of counting the alive neighbours of a cell. In our initial implementation, we iterated over all possible neighbours and used an `isAlive()` function to check if the neighbour was valid and alive. This required the use of the mod function eight times per cell. The trace displayed to us that a significant amount of time was spent in the `isAlive()` function
     – The solution to this was directly summing all possible neighbour coordinates in the grid and dividing by 255 to obtain the total count of alive cells. This works as alive cells will be set to 255 and dead cells will be set to 0. This also reduces the number of times we use modulus as we calculate the neighbouring coordinates in advance and only use modulus a maximum of two times per cell.
     – Table 6 shows the performance increase.

1) FINAL STATE IMAGE OUTPUTS:

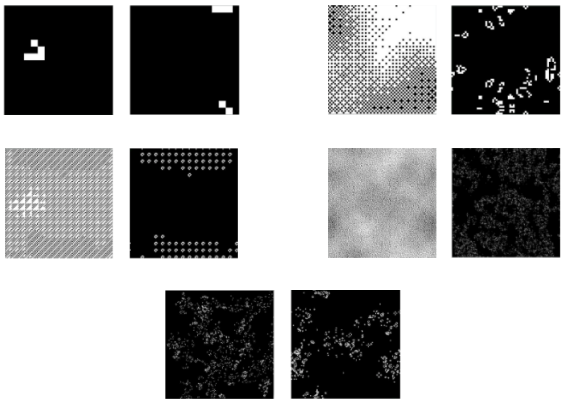

Fig. 1. Input Images with Corresponding Generated Output Images After 100 Turns
(Top left: 16x16, Top right: 64x64, Mid-left: 128x128, Mid-right:256x265, Bottom: 512x512

2) COMPARISON TO BASELINE BENCHMARKS

STAGE 1A: SEQUENTIAL :

TABLE I
BASELINE BENCHMARK COMPARISONS FOR STAGE 1A

Benchmark	Baseline Result (ns/1000 turns)	Stage 1A Result (ns/1000 turns)	Difference (%)
128x128x2-12	934401500	156421039	597%
128x128x4-12	633481008	146607627	432%
128x128x8-12	484371280	144410000	335%

STAGE 1B: CONCURRENT USING MESSAGE PASSING :

TABLE II
BASELINE BENCHMARK COMPARISONS FOR STAGE 1B

Benchmark	Baseline Result (ns/1000 turns)	Stage 1B Result (ns/1000 turns)	Difference (%)
128x128x2-12	951723879	1691069463	56%
128x128x4-12	654905004	1526687514	42%
128x128x8-12	524078916	1458421567	35%

STAGE 4: CONCURRENT USING MESSAGE PASSING AND HALO EXCHANGE :

TABLE III
BASELINE BENCHMARK COMPARISONS FOR STAGE 4

Benchmark	Baseline Result (ns/1000 turns)	Stage 4 Result (ns/1000 turns)	Difference (%)
128x128x2-12	749570815	132641552	565%
128x128x4-12	532010737	110171594	482%
128x128x8-12	362956285	97986007	370%

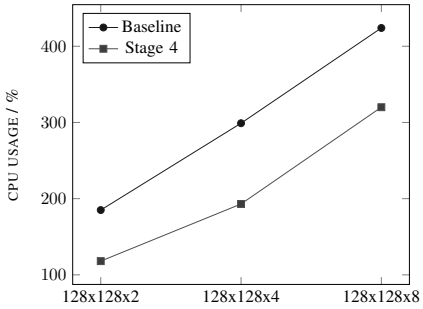

Fig. 2. Stage 4: Comparison to Baseline - CPU

STAGE 5: CONCURRENT USING SHARED MEMORY :

TABLE IV
BASELINE BENCHMARK COMPARISONS FOR STAGE 5

Benchmark	Baseline Result (ns/1000 turns)	Stage 5 Result (ns/1000 turns)	Difference (%)
128x128x2-12	962080726	104093784	924%
128x128x4-12	740846245	107814423	687%
128x128x8-12	592292244	113513630	521%

3) COMPARISON BETWEEN STAGES:

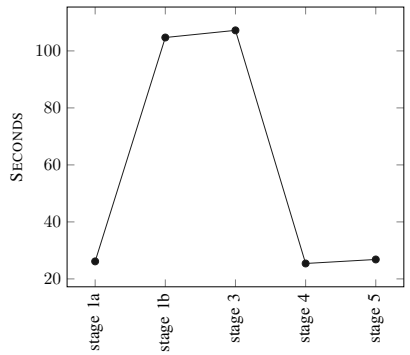

Fig. 3. Comparison Between Stages: Benchmark times per stage

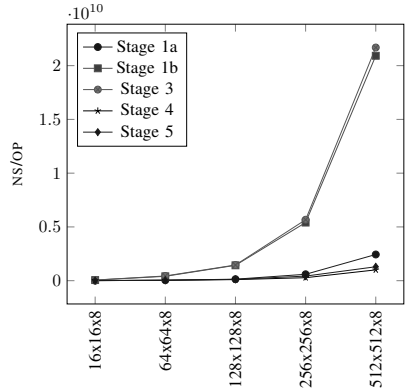

Fig. 4. Comparison Between Stage: Differing File Sizes Over 100 Turns and 8 Workers for Concurrent Implementations

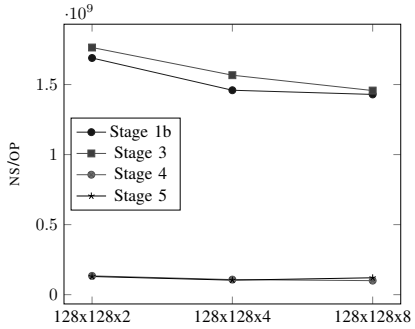

Fig. 5. Comparison Between Concurrent Implementations: 128x128 Over 100 Turns and Varying Workers

4) Optimisations Made

Distributing to Workers :

TABLE V
BENCHMARK COMPARISONS FOR STAGE 1B USING DIFFERENT METHODS OF DISTRIBUTING WORKERS

Benchmark	Synchronous (ns/1000 turns)	Asynchronous (ns/1000 turns)	Speed Increase
128x128x2-12	2168422213	1691069463	$1.2X$
128x128x4-12	2549354982	1526687514	$1.6X$
128x128x8-12	2385729074	1458421567	$1.6X$

Counting Alive Neighbours :

TABLE VI
BENCHMARK COMPARISONS FOR STAGE 4 USING DIFFERENT METHODS OF COUNTING ALIVE NEIGHBOURS

Benchmark	isAlive() (ns/1000 turns)	Improved (ns/1000 turns)	Speed Increase
128x128x2-12	848306250	132641552	$6.9X$
128x128x4-12	586077830	110171594	$5.3X$
128x128x8-12	392311250	97986007	$4X$

5) Limitations

STAGE 1A : Although stage 1a is fast, this sequential approach takes up double the memory of the concurrent implementation of stage 1b. This is because it needs two copies of the world too keep track of the current game state. When comparing the CPU usage of stage 1a against the baseline it shows a maximum usage of 100% vs 366% for the baseline. This shows the advantage of a concurrent approach utilising the potential of the CPU rather than limiting to one core.

STAGE 1B : While stage 1b implements a concurrent approach to the solution it is slower than stage 1a because the slices of the world are sent bit by bit from the distributor and received bit by bit within the worker and then when the worker sends each bit back to the distributor. This means, per turn, the world is iterated over multiple times instead of just once like in stage 1a. This also involves a lot of communication/message passing which costs efficiency in both speed and memory.

STAGE 4 : Stage 4 improves on stage 1b by no longer sending the slice to the workers at each turn instead we are sending two rows in buffered channels between the workers. A limitation still exists when sending the 'HALOS' as they must be sent bit-by-bit through the halo channels before receiving in order to prevent deadlock.

STAGE 5 : Stage 5 uses a memory sharing model meaning each worker has a pointer to the same world/modifies the same memory address. This means we must wait for all workers to finish on the current state before starting the next turn else a data race will occur. The biggest limitation is accounting for these race conditions, otherwise it performs very well.

IV CONCLUSION

Throughout the project we have focused on implementing an optimised concurrent approach to the game of life problem. In the end our fastest concurrent approach was still only as fast as the sequential approach from the first stage.

A concurrent approach intuitively seems to be a better implementation because splitting up the work means less time to compute but this is not what our findings showed.

- We implemented both sequential and concurrent methods using the memory sharing model and message passing model (halo exchange) which have similar performance in the benchmarks. The advantage of the message passing model is that it can be used across two different computers/processors whereas a memory sharing model needs to run on the same machine. We also found that sequentially solving the problem was just as effective.
- The memory sharing model was easier to implement than an optimised/efficient message passing model but requires thinking about data races.
- Stage 1b provided a concurrent solution but there was too much message passing and communication between the distributor and workers leading to bottlenecks and a much slower benchmark time compared to the sequential stage 1a. This stopped being an issue with halo exchange.

A concurrent approach doesn't always lead to the best results; its implementation needs to be optimised for concurrency before there is any potential benefit. For most applications a sequential approach may be better than a concurrent approach. For game of life, the result (for sequential, memory sharing and message passing) is the same and so sequential also seems the intuitive route because of the ease of implementation.