Concurrent Programming Report 1 - Game of Life

Jordan Taylor, George Sains

1. Introduction

This assignment involved taking various principles from concurrent computing and implement them into the Game of Life system. Throughout this project we tested and benchmarked each of the stages, testing the speed of processing different scenarios. We would then compare them to the base case which held just the logic without parallelism and use tools such as CPU profiling to examine what was bottlenecking the implementation.

2. Functionality and Design

2.1. Initial Logical Implementation:

Stage 1 involved simply creating an implementation of Game of Life, ignoring principles of parallelism and concurrency. We successfully created a serial implementation of the game's logic, utilising only a single processing thread.

Our implementation involved reading in an input board state to a 2d slice of bytes. Every turn would reconstruct the world state into a slice this way. At this point the Game of Life logic is applied.

However, we discovered that it would not be feasible to implement the logic using an in-place algorithm due to potential miscommunication between cells of different update states. As a result, the rules were applied to each cell of the game board in sequence, and the results copied to a new world structure.

2.2. Parallelisation:

The principles of concurrent computation were implemented into the program at this stage. The task of processing each turn was to be distributed between multiple workers; each of which is responsible for an equally sized board segment. The solution could not make use of memory sharing; rather goroutines should communicate by sending individual data items on demand, therefore ensuring a scalable solution.

2.3. User input & Periodic Events:

After we had implemented the parallelised version of the Game of Life, we added 3 keypresses that the user could use during runtime. This involved taking the "getKeyboardCommand" function from "control.go" and passing key presses through a channel to the distributor. The commands that we added were to pause the application, export the current state of the board to a PGM image and to export as well as stop running.

At this stage we also worked on having the program print the total number of cells which were currently alive every two seconds. This was done using a ticker goroutine running in parallel to the distributor and asking it to print through a channel.

2.4. Division of Work:

We thought a lot about how to break the problem down into non even segments. The requirement was to support all multiples of two, however we found a solution which works for any number of worker threads. The way we did this was by using floor division to work out a standard thread size, and then having the final thread take the rest of the board.

2.5. Cooperative Problem-Solving:

The final section of our design was to create a halo exchange system so as to remove the need of reconstructing the world each turn. The workers would all have two channels connecting them to the worker above & below them, which would allow them to pass the data on the edge of their segment to other workers. They would then perform the logic and reconstruct their part of the board.

After this was added, we had to get the other sections of the code working, such as the user commands and the periodic events. The commands were done using channels communicating between the distributor and the workers, with integers being sent between them to instruct different commands. The periodic events were done by having the ticker communicate with the workers in a similar fashion and then add up all the results from the workers.

3. Tests, Experiments & Critical Analysis

3.1. Initial Logical Implementation:

The benchmarking software yielded the following results for the sequential implementation. Benchmarks were run over 10 trials on MVB 2.11 lab machines. As a control measure, no other programs were running during the benchmark execution. Benchmarks were all run within the hour to minimise the differences between MVB's distributed processing load affecting the results.

Table 1: Serial benchmark results. Mean: 131.280s ± 0.745s with a variance of 0.555s and a range of 2.546s.

Trial	Benchmark Time (s)	% difference from mean
1	130.196	0.829
2	132.470	0.895
3	130.678	0.460
4	131.412	0.100
5	129.924	1.038
6	131.354	0.056
7	131.870	0.448
8	131.669	0.296
9	131.674	0.300
10	131.556	0.210

Our benchmark data set yielded a low standard deviation. All of our data points are clustered about the mean. This indicates a low variability between benchmarks; a fact further exhibited by the small range of results.

3.2. Parallelisation:

The benchmarking software yielded the following results for the parallel implementation:

Table 2: Parallel mk1 benchmark results. Mean: 327.402s ± 6.595s with a variance of 0.555s and a range of 22.046s.

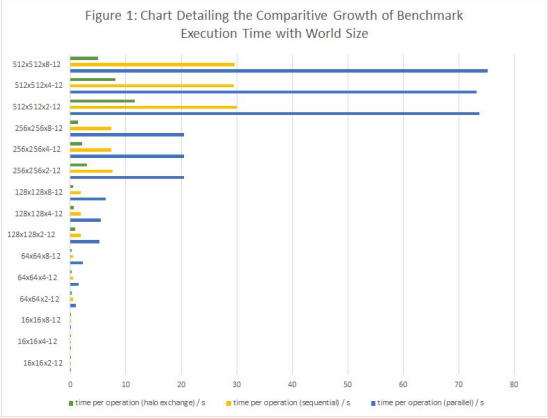
Trial	Benchmark Time (s)	% difference from mean
1	319.44	2.462
2	336.081	2.616
3	324.203	0.982
4	329.525	0.646
5	334.927	2.272
6	326.717	0.209
7	325.623	0.545
8	330.161	0.839
9	333.312	1.789
10	314.035	4.168

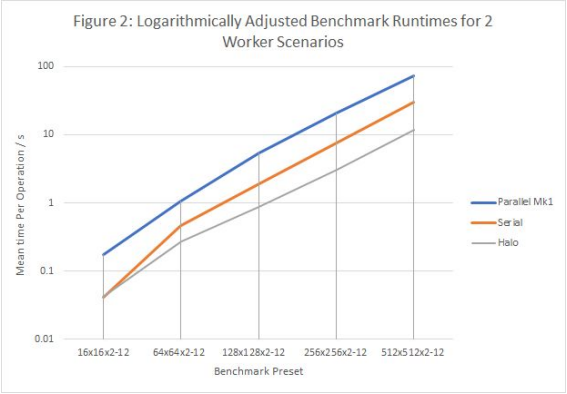
3.3. Cooperative Problem Solving:

Table 3: Parallel with halo exchange scheme benchmark results. Mean: 49.640s ± 1.256s with a variance of 1.578s and a range of 4.321s.

Trial	Benchmark Time (s)	% difference from mean
1	50.174	2.462
2	49.515	2.616
3	49.514	0.982
4	48.054	0.646
5	47.075	2.272
6	49.570	0.209
7	49.611	0.545
8	50.095	0.839
9	51.394	1.789
10	51.396	4.168

3.4. Comparison of Datasets:





The benchmarks for the first parallel implementation show an overall performance decrease compared to the serial program. The parallel benchmarks also yielded a greater range of spread. The standard deviation is 6.595, notably larger than 0.745. The mean benchmark time was 327.402s compared to the serial implementation's 131.28s. This is a time increase by a factor of 2.5.

Figure 1 displays the comparison between the serial and parallel implementation's mean execution time based on each benchmark's set of tests. For the parallelised implementations, each trial was executed with a set number of worker threads.

In the first parallel program, the number of threads has negligible impact on overall runtime. However, the size of the grid affects the runtime dramatically. Conversely, the halo exchange model manages to distribute its workload. Benchmark times noticeably decrease with the number of threads utilised.

Figure 3: An excerpt from Parallel Mk1 CPU profile

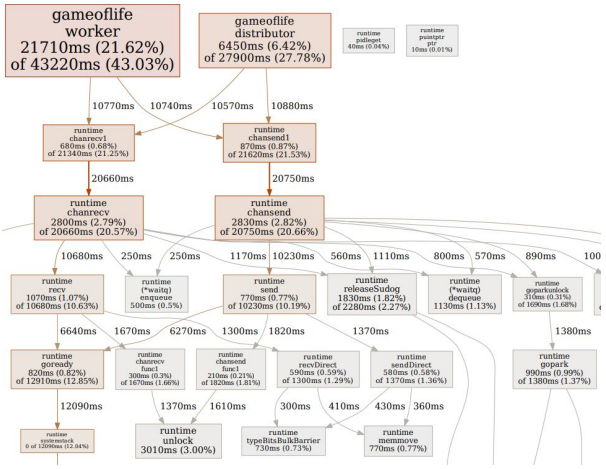
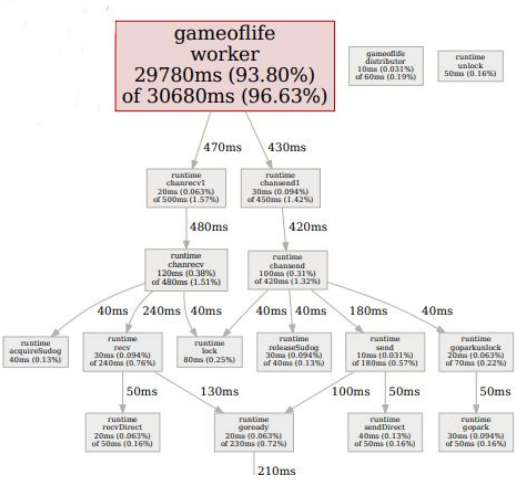
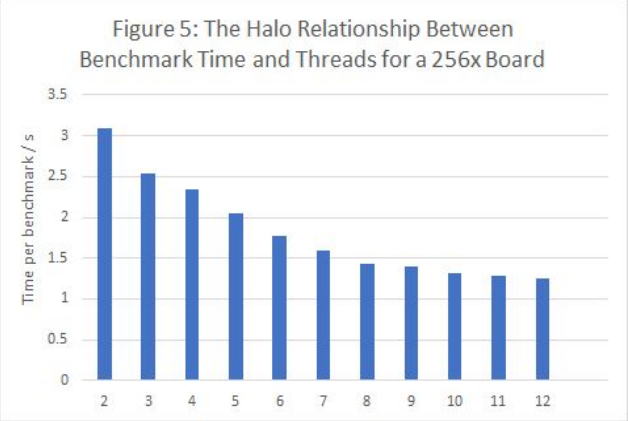


Figure 4: Excerpt from Parallel with halo exchange CPU profile



The parallelised performance decrease is due to how the world is divided and reconstructed between worker goroutines on the start and end of each turn, therefore introducing a relatively large amount of overhead. This reduces scalability because the world must be reassembled in the single threaded distributor function. This is evidenced in Figure 3, showing how the vast majority of the program's runtime is spent in a serial bottleneck communicating via channels between the distributor and various worker goroutines. Conversely, Figure 4 shows how implementing a halo exchange system mitigates the performance drop. Once the workers are called, all game logic and thread communication is handled internally. As a result, almost all of the runtime is spent processing efficiently with comparatively very little overhead.



Benchmarks were conducted on the 12 core machines in MVB 2.11. Figure 5 shows how our parallelised halo implementation can effectively make use of all physical cores. The chart exhibits a diminishing returns behaviour with regards to execution time. This is due to the associated overhead with channel communication as the number of workers increase; preventing a linear performance improvement as the thread count is increased. Further testing revealed that there is no benefit to performance when the amount of workers is greater than the number of cores. In fact, profiling showed a slight performance drop with each subsequent worker after 12.

3.5. Potential Improvements:

Our Game of Life implementation is able to handle any number of worker threads that is less than the height of the input image. It does this by assigning a number of rows to each worker equal to the floored division of the number of rows and specified thread count. The final worker takes on the remainder of rows. For some thread configurations this results in an uneven division of labour; the last worker may take on a disproportionately large load. To rectify this problem, the program could be redesigned to assign the maximum amount of rows per worker and assign the smaller remainder to the last worker.

All our implementations of GOL utilise Go's mod function to handle cell edge cases in the world structure. It has been determined that this is a CPU expensive procedure. It is called on almost every loop iteration, therefore introducing unnecessary overhead and increasing our attained benchmark times. To rectify this issue, a series of simple conditional checks could replace mod calculations in the GOL logic. Although this would extend the length of the code, it would improve its operational efficiency, resulting in reduced benchmark times.

4. Conclusion

From this assignment we have seen how large of an effect correct parallel implementation can have on the computing speed of a program. Efficient usage of parallel threads allows for over 2.5x speed of execution when compared to the standard serial implementation.

Another interesting finding was how the the initial parallel implementation was more than twice as slow as the serial implementation. This was almost entirely bottlenecked because of the way the world was reconstructed every turn and passed the bytes between workers and the distributor. The solution would have been significantly faster if memory sharing was utilised, however, that would undermine the point of a truly distributed system.