Game of Life CSA Report

Callum Ward ke19697
Lucy Randewich xm20246

Introduction

The goal of this project was to produce both parallel and distributed implementations of Conway's Game of Life, the aim of both being to divide up work efficiently between multiple workers on the same and separate CPUs respectively. We sought to produce programs which scale effectively as the number of workers n increases; our definition of scalability is reduced to comparing runtimes of implementations as $n \to max$. This report outlines and analyses differing versions of the programs against each other and discusses the algorithmic and possible hardware-related factors driving the results gathered.

Stage 1- Parallel Implementation

1. Functionality and Design

1.1 Functionality Outline

We fully completed all five steps of the Parallel Implementation Guide such that all test cases pass, and the live progress of the game is correctly displayed using SDL. As suggested in Step 1, we started with a working serial implementation of the Game of Life and extended this to evolve Game of Life using multiple worker goroutines on a single machine.

1.1.1 Serial Implementation

Considering that the program would eventually be extended to involve workers, the serial implementation was designed to use a single general function for performing a Game of Life iteration which could be called with specific parameters by workers. This function is described in more detail under section 1.2.1.

1.1.2 Extending to Parallel Implementation

The distributer uses the function $\lfloor h \div n \rfloor$ which outputs a "block length" $b \in \mathbb{Z}^+$ dependent on the height of the image h and number of worker threads n specified by the parameters of each execution. The output b specifies how many consecutive rows of the image each worker should be assigned, such that the division of work between threads is equal. Before starting the workers to execute a turn of the game, an immutable matrix is created and passed to each worker goroutine. This way, goroutines will not be able to interfere with the matrix that other routines are working on and there will be no undefined behaviour. This operation is understandably memory-heavy since each worker has its own version of the entire image in memory; section 1.2.2 outlines overheads and improvements to this.

1.1.3 SDL and Event Handling

Aside from workers, there are two other goroutines involved in the execution process. One defines ticker functionality and the other handles keypresses from the SDL window; both functions are initiated in the distributer and communicate via channels. The function "processKeyPresses" uses a switch statement to listen on the keyPresses channel and send interrupts via channels back to the distributer accordingly. The distributer listens on these channels at the beginning of each turn of the game, and only proceeds to process the turn if it receives no interrupts and the default statement is reached. The distributer also communicates with the I/O process via channels when required i.e., for input and output of boards.

1.2 Efficient Design

1.2.1 Serial Implementation Efficiency

We then had to decide between two different methods of implementing the Game of Life logic; one concept was to create an array of neighbouring cells, accounting for image boarders using sequential if statements, then count how many of these neighbouring cells had value 255 and alter the world accordingly (later referred to as Version 1). We instead chose to use our other design (Version 2), which reduced runtime as well as memory usage by going through each neighbouring cell and incrementing a counter by each of their values, using modulus functions to account for image boarders. It then divides the count by 255 to hold the number of "alive" neighbouring cells, and proceeds like the initial design to alter the world accordingly. Section 2.2 explores in more depth the differences between these implementations.

1.2.2 Parallel Implementation Efficiency

Using the algorithm outlined in section 1.1.2, the work for executing a turn of the game is divided up equally between workers. However, for some combination of number of threads n and height of board h, it will be the case that $h \div n \notin \mathbb{Z}$. To accommodate this, the function uses a floor operation to choose the lower bound of the division and the final worker is given the remainder of the board to evolve. The drawback of this is that one worker might take longer to process its data than the rest, which results in the distributer blocking until this single routine has finished.

We have written an alternate solution in which each worker only creates and returns via a channel the specific region of data it has been told to work on by the distributer, which collects the slices together by using an append function once each worker finishes its task. The differences in memory use and runtime between this and the original implementation are explored in section 2.3.

2. Critical analysis

2.1 Analysis Outline

In this section we will provide a selection of results gathered from our benchmarking analysis. The command "go test -run ^$ -bench . -benchtime 1x -count 10 | tee results.out" was used to run the benchmarking function in distributer_test.go, such that each sub-benchmark is run 10 times and reported individually. Each test was run on the same 8-core processor to simulate the same conditions to allow for fair comparison of test data, and each test used the 512x512 image with the number of turns specified as 500. This number is large enough to provide an adequate representation of the scalability of the different implementations, since the difference between runtimes is more apparent with a larger number of turns of the game.

2.2 Alternate Serial Implementation Analysis

Figure 1 shows the difference in runtimes of both the serial implementations outlined in 1.2.1. Version 2 has on average a very significant 29.6% decrease on the runtime of version 1. This result confirms the assumption made earlier that the modulus function method of accounting for image boarders far more efficient. Another property to note from the result set is how the variance of version 2 is negligible compared to that of version 1. While this was not a result we had anticipated, it gives another reason to conclude that version 2 is the better choice of algorithm as the runtime is more reliable. By looking at the differences in both function versions, it would be intuitive to draw the conclusion that memory allocation in Go is not consistent in terms of runtimes. One possible explanation for this could be that in version 2 there is enough space for all memory to be allocated on the stack which only involves two CPU instructions, while in some trials of version 1 dynamic allocation onto the heap may be occurring at runtime.

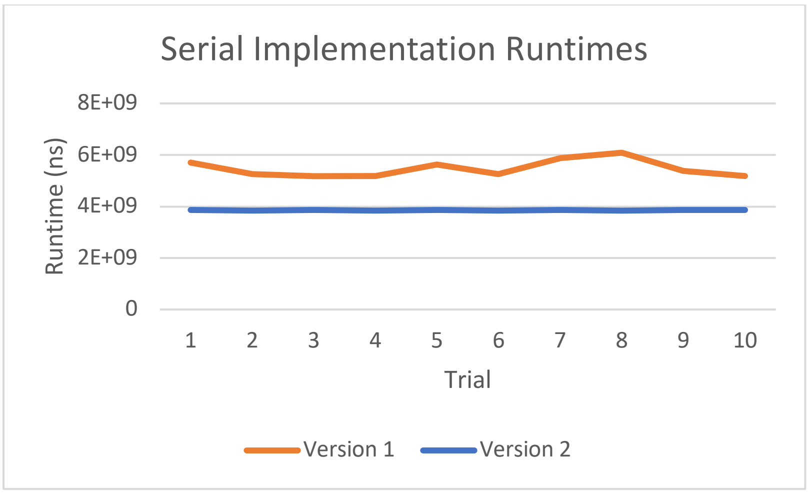

Fig. 1

2.3 Serial vs Parallel Analysis

The following graph (figure 2) is the result of benchmarking our parallel implementation. This pattern is expected, showing that the program runs faster when more worker threads are added. The largest difference in runtime is between one and two worker threads, with the percentage change decreasing in a reciprocal fashion as the number of threads doubles. This is due to the overheads associated with adding more goroutines, for example each routine must allocate its own memory. For this reason, when each worker only has a small task to perform due to a high number of workers, the overheads become more predominant in runtime determination and the time saved in parallelism becomes negligible in comparison.

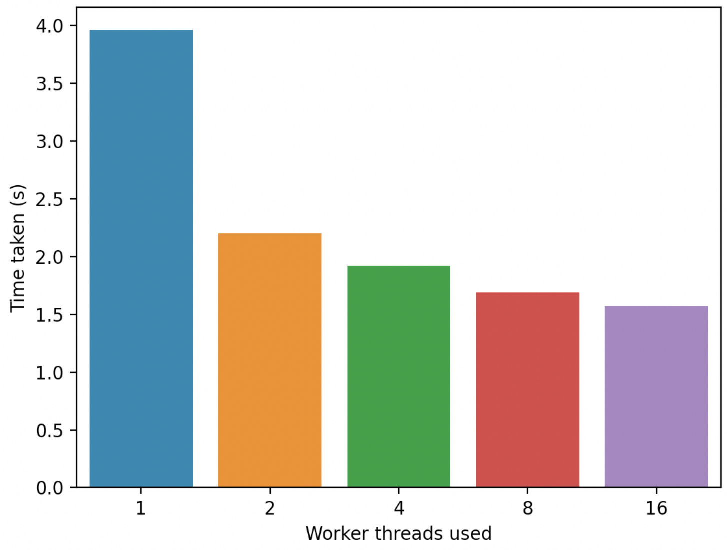

Fig. 2

Figure 3 below depicts the differences in runtime between the serial implementation (version 2) and the parallel. As can be seen at the y-intercept of the graph, the serial implementation is slightly faster than the parallel implementation with a single worker thread; this is due to the extra goroutine call and channel operations associated with the parallel implementation. However, for subsequent numbers of workers the runtime of the parallel implementation proceeds to cut that of the serial implementation in half.

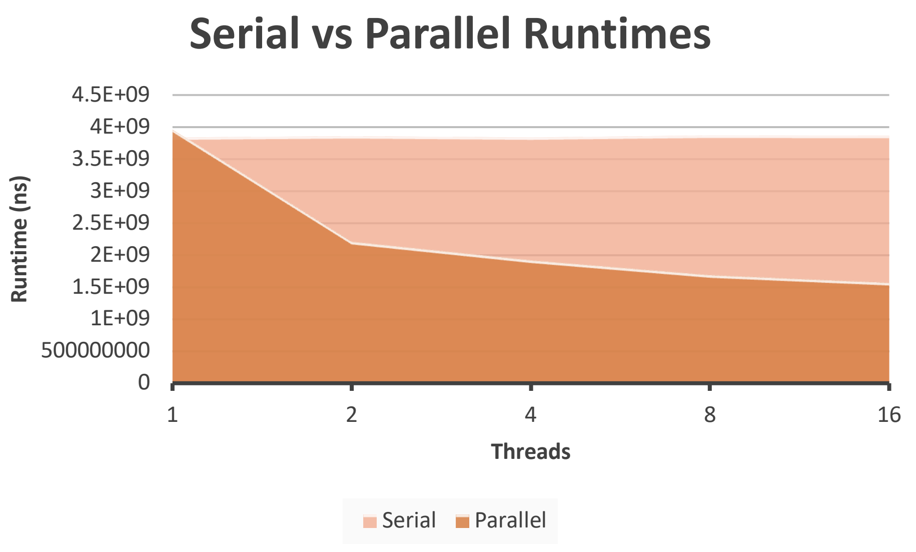

Fig. 3

2.4 Alternate Parallel Implementation Analysis

After benchmarking the alternate solution described in section 1.2.2, it was evident that our initial implementation was more efficient. Evidently, the time taken in appending the slices back together prevailed over the time saved in the workers only having to create slices for the data they produced. The graph comparing their runtimes below (Figure 4) brings to light an interesting point, however; the runtime of version 2 appears to curve back upwards after 8 workers, hinting at a possible a parabolic relationship rather than reciprocal.

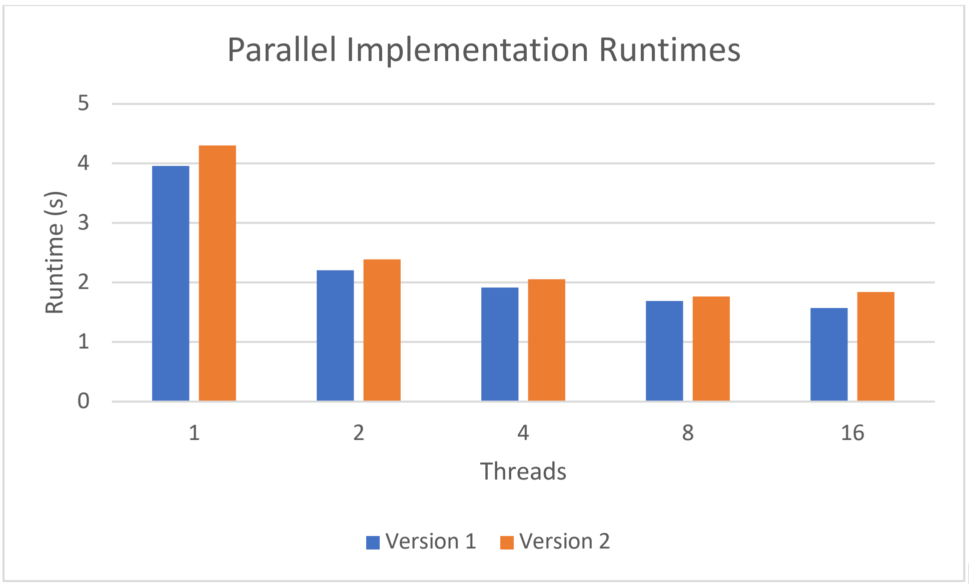

Fig. 4

This discovery led us to think about how our initial parallel implementation would scale when more than 16 workers are used. It became evident to us that when the number of workers $n \mid 1 \leq n \leq \mathit{image\ height}$ surpasses the number of logical cores in the CPU used for benchmarking (in our case 8), the solution no longer lends itself directly to the number of physical cores available and the OS must use logical cores among other methods beyond the extent of this report to accommodate for the number of threads required by the program. The parabolic nature of Figure 5 supports this claim.

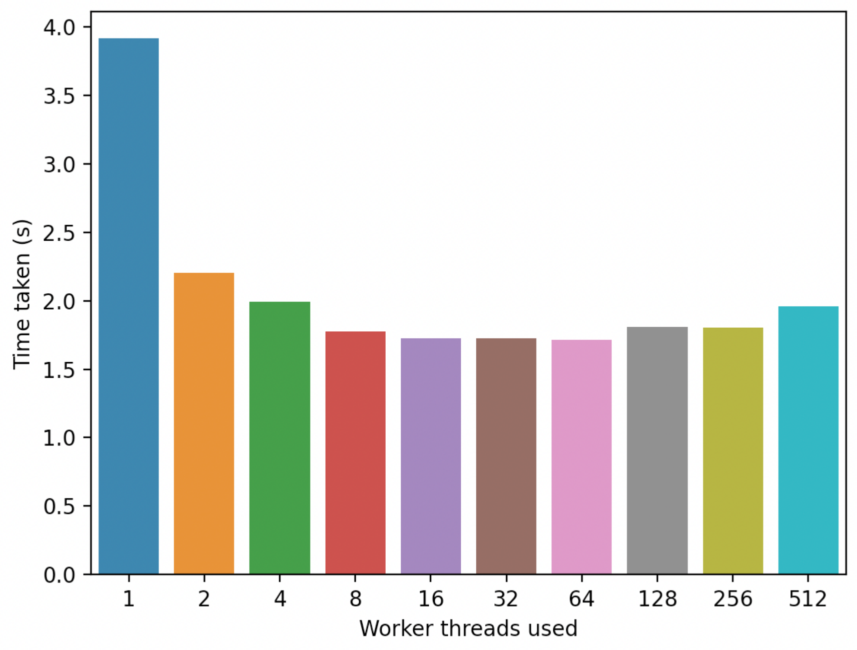

Fig. 5

3. Conclusion

3.1 Possible Improvements

If we had more time for the project, it would have been interesting to use execution profiling to determine which parts of the program dominated the runtime. This data could have pointed to areas of our code which were less efficient, so that we could have considered optimisations for these areas to reduce overall runtime. One of these areas we have been able to pinpoint was that we chose to create workers and assign them parts of the world at the start of each turn. Had they been set up before the game began, both space and time complexity could have been reduced.

Stage 2- Distributed Implementation

1. Functionality and Design

1.1 Distributed System Design

We first built a system which split the components up into one client and one server, which passed all tests including SDL keyboard input which we were able to test by running the server locally. We then moved on to creating a scalable distributed implementation which uses a broker and "halo exchange" to divide work between worker nodes each running on a separate AWS instance.
The system comprises of three main components: the distributor, the broker, and worker nodes. When the broker is run, worker nodes can dial into it using its private IP. Each worker uses two command line arguments to run, the first being its own IP and the second being the IP of the broker. Once all workers have registered, the broker allocates each worker a slice of the world when the distributor calls the broker with a world to be processed. Each worker then processes the turns of the game, altering its own data and retrieving "halo regions" from its neighbouring workers. This data exchange blocks until all workers have finished processing the turn, so that they can then move on to processing the next turn together. Once all turns are completed, a response is sent back to the broker which appends the world slices back together and sends the final world back to the distributor.
We had some difficulty in setting up the alive cells count in this version of the distributed system, since workers sometimes received the call from the broker to retrieve a cell count when one worker may have been one turn further in its computation than another. This resulted in a mismatch of cell counts being returned. To combat this, we made each worker store the number of alive cells after each turn in an array. The first worker to get the alive cells

call sends the turn in its response which is then used by each subsequent worker, which all return the correct number of cells which were alive on the turn specified by the first worker.

2. Critical analysis

2.1 Analysis Outline

We ran a benchmarking function on the distributor to produce a graph showing how the performance of our distributed system scales with the number of worker nodes.

2.2 System Scalability

The graph below (figure 6) was created using the large 5120x5120pgm image, with each benchmark test performing 30 turns of the game. Each value was calculated to be the average of 10 iterations of the benchmarking test. Due to the small trial size, it is unclear whether the relationship between data points is reciprocal or linear. It is clear however that the system scales well with the addition of more workers despite the added overheads of halo communication between workers. Since each worker is on its own c4.XLarge AWS instance, the system can exploit 4 cores per worker. Although we are not explicitly dividing work between each core on each node, the Golang runtime uses time-sharing to optimise computation between all nodes available.

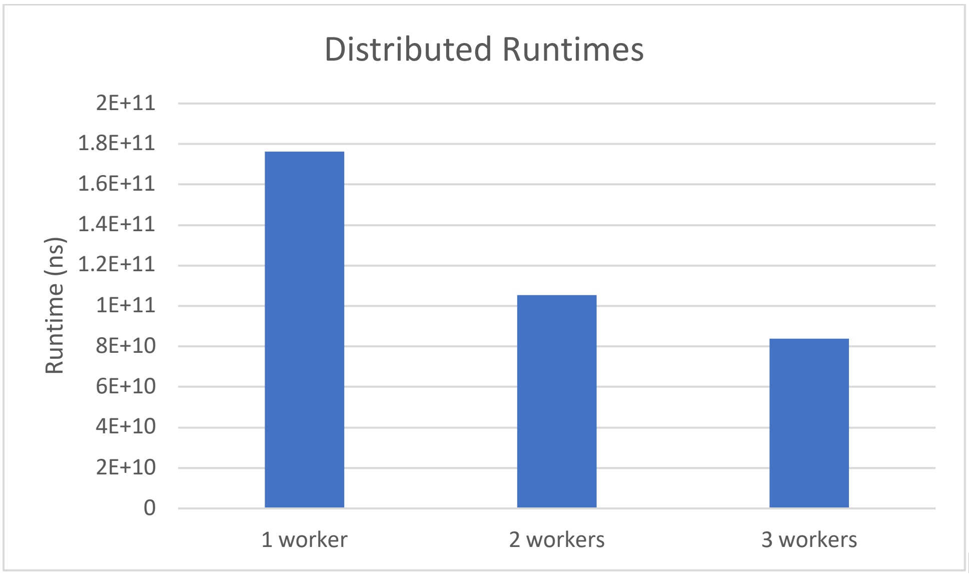

Fig. 6

3. Conclusion

3.1 Possible Improvements

We would have liked to combine our parallel and distributed implementations, such that each worker node would use parallelisation to compute the iteration of its part of the board. This would considerably increase the scalability of the system, perhaps to the extent of overtaking our parallel implementation, due to being able to exploit the use of each core of the CPUs on each worker node. For example, if there were three nodes each with 8-core processors, there would be 24 physical cores for the system to exploit and almost certainly more than 24 logical threads.
We also are aware that our system does not have good stability, such that if one element loses connection or experiences an error it will cause other elements of the system to either crash or block forever. Had we had more time for the project we would have liked to put error handling systems in place to ensure graceful shutdown of every element in any eventuality.