Game of Life – Report

Valeriu-Andrei Florescu (la18889), Daniel Salter (jj18461)

Firstly, after having read the tasks assigned to us, we began implementing the game logic for stage 1A. This required an understanding of the rules of the game, for which we have prepared by doing a bit of research and watching videos of how the program behaves. We have added the game logic in the distributor function, inside a loop that runs "p.turns" times. Here, for each cell in the "world" we counted the number of neighbouring cells that are alive. Once the logic has been carried out for every turn, the world is sent to "pgm" one byte at a time, which saves a "pgm" representation of the world inside the "out" folder. We used this and the tests to check the correctness of our game logic.

When implementing a divide and conquer paradigm, the first obstacle we encountered was how to send the required 'halos' to each worker. A small modification to the section of code that sends slices of the world to respective workers introduced the sending of rows above and below the slice, known as halos. This then brought to light the nuance that the world effectively loops back over from the top to the bottom (and vice versa). For example, the first worker's top halo is the bottom row of the world. As things were, the code would throw "index out of bounds" errors as the first worker indexed the -1th row as its top halo. The addition of a modulo calculation enabled this to instead index the last row of the world.

Moreover, when implementing the key press handler, we implemented a select statement to tackle the cases when relevant keys are pressed. The program terminates and generates a final "pgm" when 'q' is pressed, it saves a most up to date "pgm" image of the current state when 's' is pressed and pauses the program when 'p' is pressed. We also moved the game logic to the default case, which makes the program run normally when no key is pressed.

Then, when we implemented the ticker, we created an anonymous function and ran it as a goroutine. This function contains a select statement that waits for a ticker to send on its channel, which happens at regular 2 second intervals, then prints the number of alive cells. This was a challenge as when pausing the program, the ticker kept printing the number of alive cells (which didn't change whilst paused). To solve this, we included a new boolean variable, pause, and made the ticker only print the number of alive cells when "pause == false". The value of pause is updated in the select statement where we have handled the key presses.

Enabling the program to use workers numbered in all multiples of 2 threw up the problem that the number of cells in the world would no longer always divide evenly between the workers. Overcoming this required some simple mathematical calculations. Once we decided to divide the work by strategically allocating the number of rows a worker will process, it became a challenge of using division and modulo operators between the total number of rows and number of worker threads to calculate exactly how many rows each worker should be allocated.

Implementing the halo exchange model provided one major issue: if every worker simultaneously first attempts to send the required bottom-halo to the worker above it then they will all lock, as no workers will be receiving the halo being sent. To overcome this, we made even numbered workers send their halos first while odd workers receive these halos being sent, then the workers switch so odd workers send and even receive.

Note that for all the data we have collected, we have taken repeat readings (either 5 or 3) from which we have calculated an average. This is to find any anomalies that could happen, and to make sure that our results are accurate and reliable.

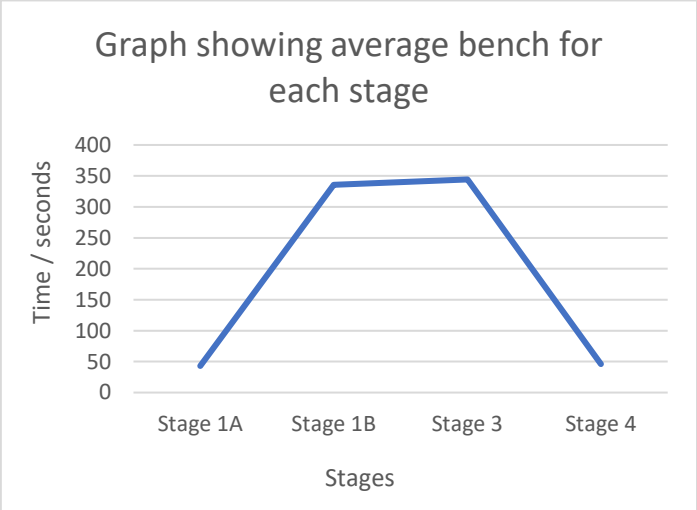
Analysis

Bench time / seconds				
Readings Taken	Stage 1A	Stage 1B	Stage 3	Stage 4
Reading 1	44.016	328.065	347.395	46.191
Reading 2	42.998	329.428	332.125	47.097
Reading 3	44.161	344.139	325.570	45.795
Reading 4	41.754	337.791	356.611	46.226
Reading 5	42.098	338.761	359.657	43.628

Above we have a table showing bench results at each stage. These 3 tables are labelled with the stage they're showing results for. We have got the results using "make bench" at each stage.

- For Stage 1A, the average bench time is 43.01s;
- For Stage 1B, the average bench time is 335.64s;
- For Stage 3, the average bench time is 344.27s;
- For Stage 4, the average bench time is 45.79s;

We will now represent this difference in a graph, and then discuss the reason behind this data.



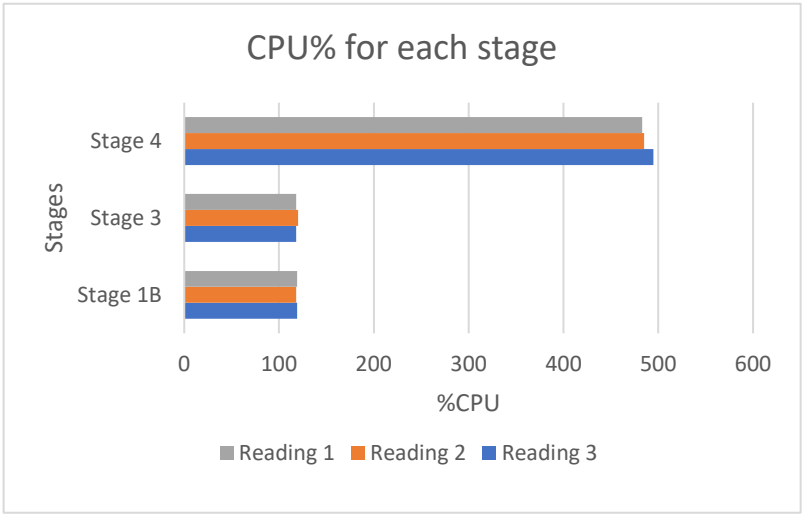
In this graph, we have plotted the average bench time at each stage. As you can see, the average bench time at stage 1A, where the program does not run in parallel, is lower compared to the bench time at stage 1B and 3. To explain the reason behind this, we ran "make time" in the console and took the CPU% data. If this is above 100%, then it shows parallelism.

It is also clear that in Stage 4, our program's run time has improved significantly compared to the previous stages (1B and 3).

After running "make time" 3 times for 1B, the CPU% average rounded up to 119%. Having looked at the System Monitor whilst running "make bench", we have made the conclusion that the program was using about 10% of each core. This does mean that the program runs in parallel, however it is using a very small percentage of each core, making it very slow. Whilst for 1A, the CPU% was 95%, which shows that 1A is using a way higher percentage of the cores, since it is not concurrent and hence does not use all the cores.

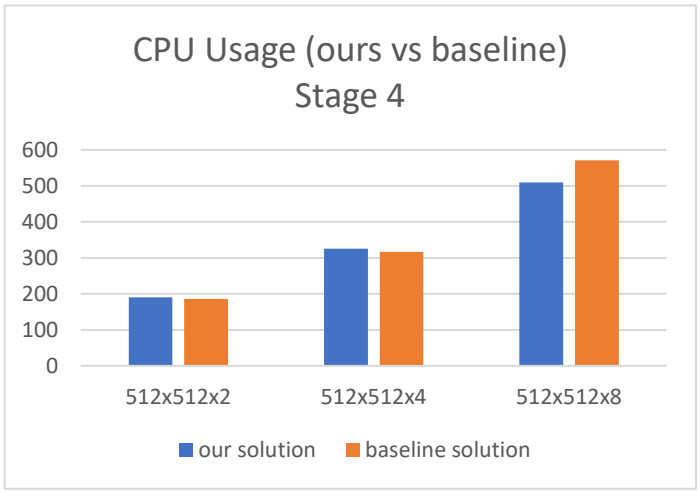
For stage 3, the CPU% was very similar to stage 1B, as it also rounds up to 119%, and having looked at the System Monitor, it was a very similar activity to the one in 1B. This is reflected in the tables above, where we can see that the difference in the average bench time between stage 1B and 3 isn't significant.

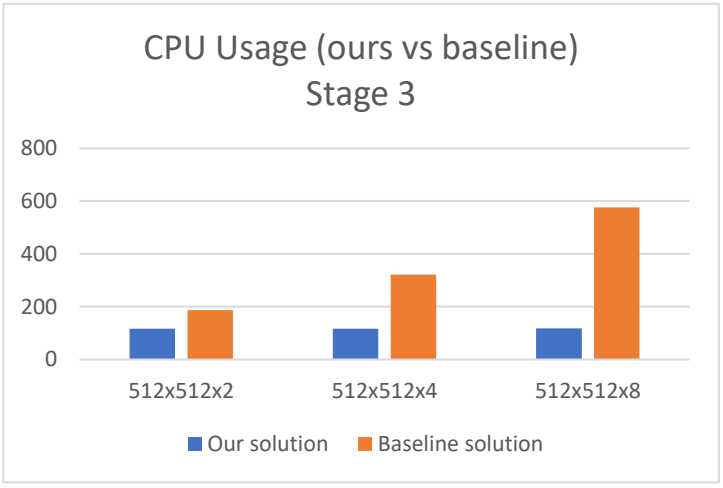
However, for stage 4, this CPU% increases to an average of 488%, which shows that a larger percentage of each core is used, which could also be seen whilst looking at the System Monitor. This shows the program runs in parallel, and it also made it faster as the cores were working harder. All this data is represented in the graph below:



We haven't taken data for stage 2 as it hasn't had a significant impact on the bench times/ CPU%. Also, in the graph on the left, we haven't included Stage 1A as it was not a concurrent method, and hence the CPU% is not important for our analysis.

Benchmarking our solution v baseline solution (512x512) - **Stage 3 vs Stage 4**



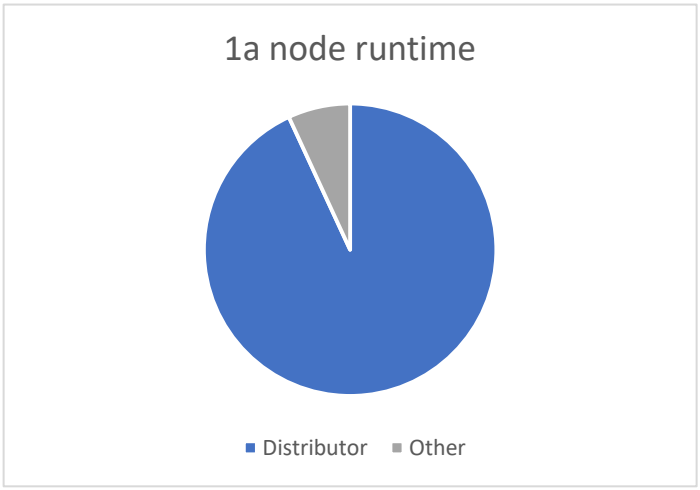


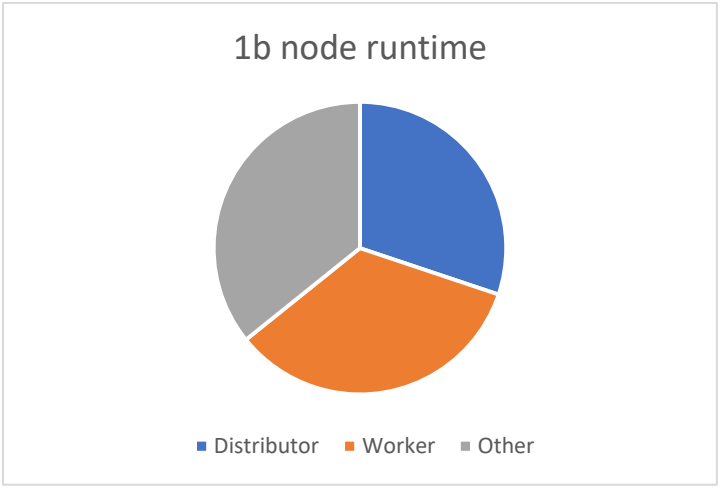
Compared to the baseline results of the benchmark when doing "make compare", our results from stage 4 show a higher CPU usage for 2 and 4 threads, although not significantly higher, and this is lower when the benchmarks is done for 8 threads. The % difference is 97%, 94% and 113% respectively for 2, 4 and 8 threads, which shows that our results are good since the smaller this percentage, the better.

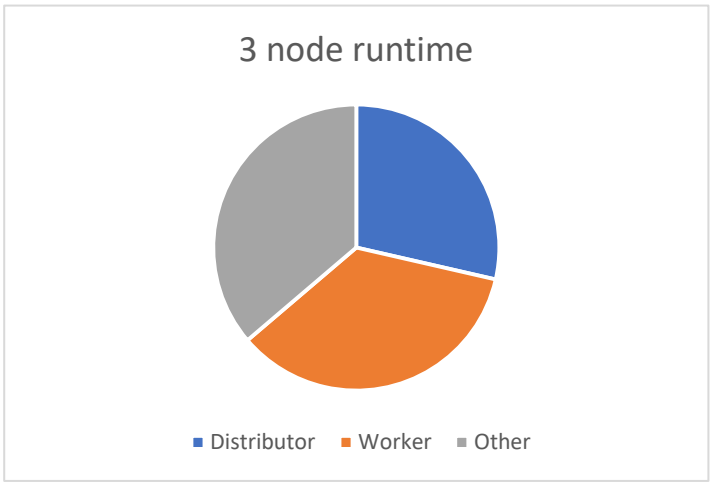
Now, looking at the results for stage 3 in comparison to the baseline, we can see that these results are not anywhere as good as the ones from stage 4. In the bar chart, we can see that the CPU usage in the baseline benchmarks is a lot higher than our results. Also, the percentage difference is 159%, 274% and 488% which shows that our stage 3 implementation gives worse results that the baseline and the stage 4 implementation.

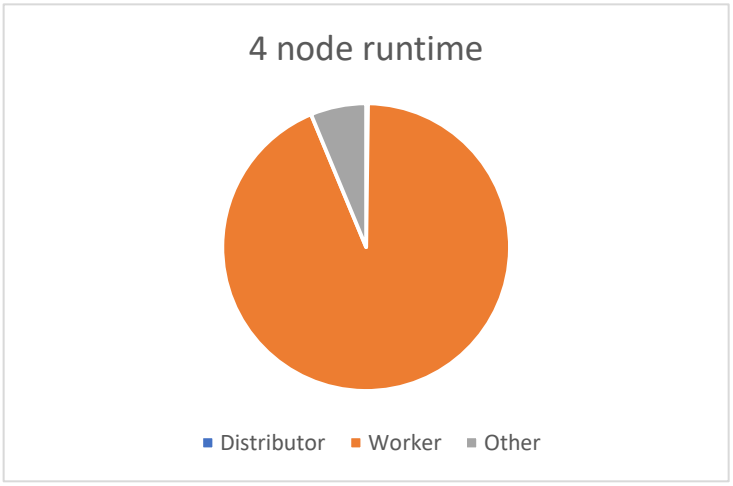
The fact that our stage 4 implementation is a significant improvement compared to our stage 3 is backed by the CPU% bar chart, and by the fact that it uses about 40% of each CPU, making the program run about 8X faster, which is shown in the bench times.

CPU profile Analysis









A first very clear observation is the similarity between the runtimes of the relevant functions in stage 1b and 3. Considering the code in these two stages this makes sense; the distinguishing features between them are user interaction, the ticker thread that prints the number of cells currently alive, and enabling the number of workers to equal any multiple of two. Each of these additions does not significantly affect the proportions in which workload is shared between distributor and worker functions.

From stage 1a to 1b the introduction of a worker function spreads the workload between the two functions evenly. The large increase in the proportion of time spent waiting for the 'other' category in 1b is accounted for by the time workers and the distributor spent locked, waiting for channels to send/receive. 70% of the time worker is running for, it is either waiting to send or receive on a channel. Similarly, this accounts for 84% of distributor's time.

The chart from 4 makes it clear that virtually no time is spent running the distributor as all the processing originally done by it has been shifted to the workers. Furthermore, the time spent waiting for channels has been significantly reduced, only 5% of the worker's time is now spent waiting.