outine uses a channel to close the keypress and ticker goroutines cleanly.

The keypress goroutine continuously checks for keypresses. When ‘q’ is pressed the respective ‘quit’ flag is set to True. The same thing happens when ‘s’ is pressed for its respective ‘savePGM’ flag, and similarly, when ‘p’ is pressed, its respective ‘pause’ flag is flipped. The execute function is responsible for executing all turns in the game, and continuously checks the ‘quit’, ‘savePGM’ and ‘pause’ flags at the start of each turn. If the quit flag is detected the execute function returns. If the ‘savePGM’ flag is detected the current world state is outputted alongside a filename string which contains the size of the world and number of turns completed. When the ‘pause’ flag is detected, a pause function is run serially which continuously checks the ‘quit’ and ‘pause’ flags, which are required for the function to return. Once the pause function returns, the execute function checks the ‘pause’ flag before the ‘quit’ flag, so the game will either quit or continue depending which key was pressed causing the pause function to return. The pause function also continously checks the ’savePGM’ flag to save the curent world state as a PGM.

The ticker loop sets the ’getAliveCount’ flag to True every 2 seconds. The execute function checks this flag at the end of each turn. If the flag is True, the number of alive cells is outputted and the flag is set to False.

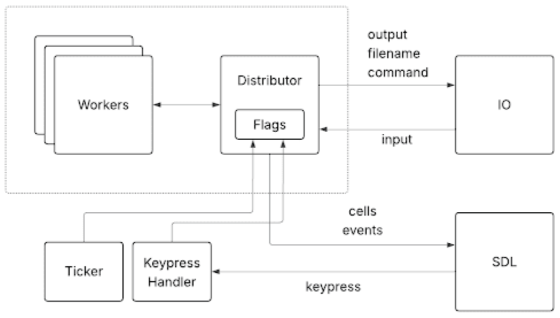


Figure 3: Concurrent Implementation Diagram

3.2 Scalability

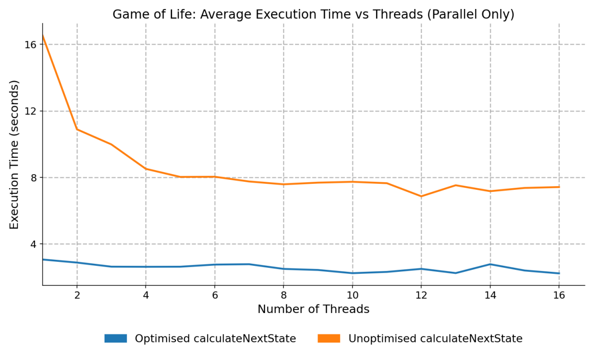
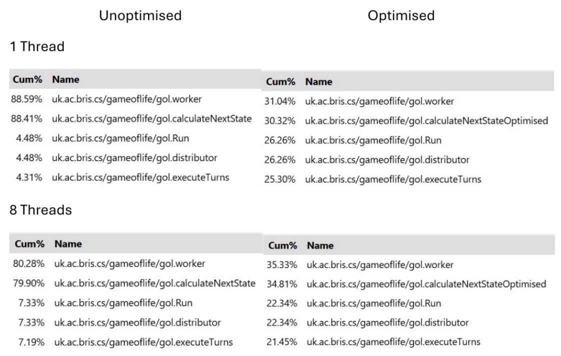


Figure 4: Initial scaling benchmark for Initial and Optimised Algorithms for Calculating Next State

Figure 4 shows how total runtime scales with the number of workers used. Using the unoptimised algorithm, increasing the number of workers initially reduces time but has diminishing gains with increasing workers. This is consistent with Amdahl’s law, since performance increase becomes more limited by serial processes with a higher worker count, particularly communication overhead and IO execution performed by the main goroutine.

The optimised version shows worse scaling. The pprof results in Figure 5 reveal that the main goroutine now accounts for a far greater portion of total runtime, demonstrating that time reduction is being mostly limited by the remaining serial processes on the main goroutine.



Unoptimised

Optimised

1 Thread

Cum%	Name	Cum%	Name
88.59%	uk.ac.bris.cs/gameoflife/gol.worker	31.04%	uk.ac.bris.cs/gameoflife/gol.worker
88.41%	uk.ac.bris.cs/gameoflife/gol.calculateNextState	30.32%	uk.ac.bris.cs/gameoflife/gol.calculateNextStateOptimised
4.48%	uk.ac.bris.cs/gameoflife/gol.Run	26.26%	uk.ac.bris.cs/gameoflife/gol.Run
4.48%	uk.ac.bris.cs/gameoflife/gol.distributor	26.26%	uk.ac.bris.cs/gameoflife/gol.distributor
4.31%	uk.ac.bris.cs/gameoflife/gol.executeTurns	25.30%	uk.ac.bris.cs/gameoflife/gol.executeTurns

8 Threads

Cum%	Name	Cum%	Name
80.28%	uk.ac.bris.cs/gameoflife/gol.worker	35.33%	uk.ac.bris.cs/gameoflife/gol.worker
79.90%	uk.ac.bris.cs/gameoflife/gol.calculateNextState	34.81%	uk.ac.bris.cs/gameoflife/gol.calculateNextStateOptimised
7.33%	uk.ac.bris.cs/gameoflife/gol.Run	22.34%	uk.ac.bris.cs/gameoflife/gol.Run
7.33%	uk.ac.bris.cs/gameoflife/gol.distributor	22.34%	uk.ac.bris.cs/gameoflife/gol.distributor
7.19%	uk.ac.bris.cs/gameoflife/gol.executeTurns	21.45%	uk.ac.bris.cs/gameoflife/gol.executeTurns

Figure 5: Pprof Analysis of Unoptimised vs Optimised Calculate Next State of Concurrent Implementation

4 Concurrent Optimisations

4.1 Modulus Operator

To handle edge-wrapping, our initial approach was to use modulus in our algorithm:

Algorithm 3: Optimised Neighbour Adding using If Statements for Edge Wrapping

```
1: **if** $w_{ij} = 255$ **then**
2:     **for** $dy = -1$ to 1 **do**
3:         **for** $dx = -1$ to 1 **do**
4:             **if** $dx = 0$ **and** $dy = 0$ **then**
5:                 **continue**
6:             **end if**
7:             $ny \leftarrow (y + dy + n)\%n$
8:             $nx \leftarrow (x + dx + m)\%m$
9:             $aliveCells[ny][nx] \leftarrow aliveCells[ny][nx] + 1$
10:        **end for**
11:    **end for**
12: **end if**
```

Another way to handle edge-wrapping is by using if statements to check if the number is out of bounds and set it to its desired value if it is, as shown in the following algorithm: