Algorithm 4: Optimised Neighbour Adding using If Statements for Edge Wrapping

```
if w_ij = 255 then
    for dy = −1 to 1 do
        for dx = −1 to 1 do
            if dx = 0 and dy = 0 then
                continue
            end if
            ny ← y + dy
            if ny < 0 or ny ≥ size then
                continue
            end if
            nx ← x + dx
            if nx < 0 then
                nx ← imageWidth − 1
            else if nx ≥ imageWidth then
                nx ← 0
            end if
            aliveCells[ny][nx] ← aliveCells[ny][nx] + 1
        end for
    end for
end if
```

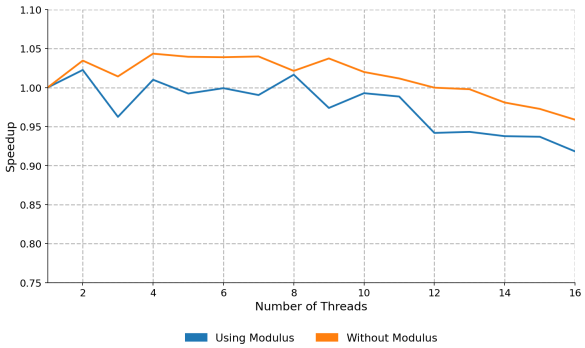

Figure 6: Using Modulus Operator vs Using If statements

Figure 6 shows the speedup that occurs when using algorithm 4 compared to 3 (execution time for 1 thread / execution time for N threads). Using if statements is resulted in a slgiht performance increase over using modulus, and that it has slightly better speedup, meaning it scales better with more threads used.

4.2 Worker Communication

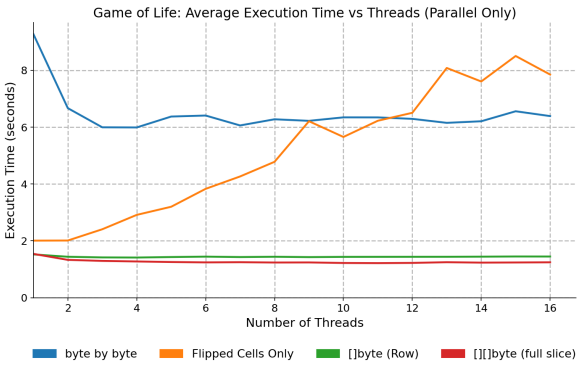

Figure 7: Worker Communication

Figure 7 shows how different ways of sending information through channels back to the main goroutine from workers affect the execution time. Our initial implementation sent a full slice of the section of the world the worker has calculated. We decided to benchmark different ways of sending back the new world state to see if we can optimise worker communication. We tested sending information in individual rows through a buffered channel immediately after each one has been calculated by the worker, and sending it byte by byte for each cell. Our results show a slight performance decrease for sending by row and a significant decrease for sending byte by byte. Although this reduces some overhead in the main goroutine as it was quicker to reconstruct the next world state, there was an overall time increase from using more channels, this indicated that sending small chunks of data through channels multiple times was likely slower than sending all the data at once chunk through a channel only once.

Seeing as there was no performance increase, we took a different approach and decided to adapt the rule set of calculating each next cell so that we can send only cells that are flipped back to the main goroutine.

$$f(w_{ij}, n) = \begin{cases} 0, & \text{if } w_{ij} = 255 \text{ and } (n < 2 \text{ or } n > 3), \\ 255, & \text{if } w_{ij} = 0 \text{ and } n = 3, \\ w_{ij}, & \text{otherwise.} \end{cases}$$

Figure 8: Modified Rule Set

This resulted in a large performance increase to the byte by byte implementation for a smaller number of workers, yet scaled poorly with increasing workers, and was still slower than our initial implementation. We speculate, this is due to increased queueing on the channel, as channel sends become far more frequent than the rate the main thread can receive with increasing workers.

Overall, our experiments showed us that increasing the number of messages sent on channels had a larger effect on time than reducing overhead in the main goroutine, therefore, sending full slices remained our strategy.

4.3 Worker Communication II

Another way of reducing worker communication overhead is to send each worker only the part of the world it needs, which is its assigned section and one extra row above and below for neighbours calculations. To implement this we used halo exchange, where each worker sent its top and bottom rows to its neighbours via channels, calculated the next state of the section of its input that excludes the top and bottom rows, then calculated the next state for the top and bottom rows after receiving the halo regions from its neighbours.