to and calls each worker to calculate a section of the next world state. After all turns are complete, the final world state is sent back to the local controller. The local controller's ticker loop communicates with the broker to receive the current world's alive cells count every 2 seconds. The client continuously checks for keypresses and communicates relevant information with the broker to handle each key press case.

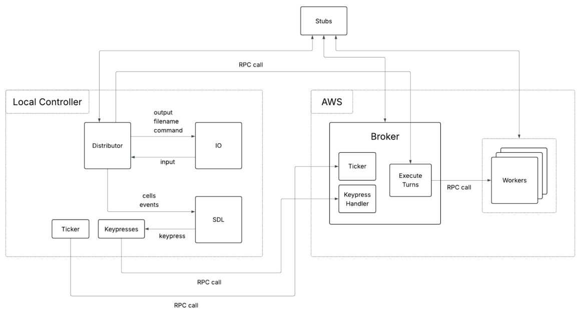

Figure 13: Distributed Implementation Diagram

## 5.2 Network Data Exchange

Here we explain what information is sent between RPC calls between the client and the broker, and the broker and workers. When calling the function to execute all turns of Gol in the broker, the local controller sends the current state of the world, number of turns to complete, and the image height and width. After executing every turn, or if the local controller signals the broker to quit, the final world state, turns complete, and slice of alive cells, is sent to the local controller. Once one of four keypresses are detected by the local controller, it sends the key to the broker:

- 'p': the broker pauses or resumes turn execution and sends whether the game is paused or resumed in the response and the number of completed turns.
- 's': the broker sends a slice of the entire current world state and number of completed turns in the response for the local controller to output a PGM image.
- 'q': the broker finishes executing the current turn, and returns, the final game state information as outlined earlier.
- 'k': the broker finishes executing the current turn, returns the final game state information as outlined earlier, signals all workers to quit, and then quits.

Every 2 seconds the local controller calls a function on the broker which sends the current number of alive cells back to the client. To calculate the next world state, the broker sends each worker a slice containing the current world state, the image width, the starting point of the world each worker must calculate from and the size of the section of the world that it needs to compute. The worker returns the new state of the section it has computed to the broker.

## 5.3 Scalability

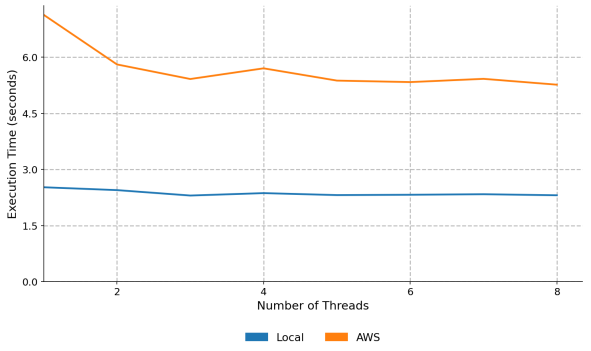

Figure 14: Scaling benchmark: AWS vs local

We benchmarked scalability for our initial distributed version locally and on AWS. For the AWS benchmark, we ran one EC2 instance for the broker, and each worker on separate EC2 instances. Figure 14 shows that, as expected, the distributed version runs slower on AWS, and with both slower than our concurrent version, due to the additional communication overhead from RPC calls. Similar to our concurrent version, we see diminishing returns with increasing workers due to RPC communication overhead, suggesting that performance would further flatten out if we added more workers, although we speculate that scaling could be improved for high worker counts, with the addition of a separate node for collecting and merging worker sections, to parallelise this process from the broker.

## 5.4 SDL Visualisation

We implemented two versions of SDL visualisation, one that calculates flipped cells in the broker and makes an RPC call to the local controller, to send them as a slice, which is then sent to IO using the events channel. The other uses the same modified ruleset from [4.2] (Figure 8), so that minimal computation is needed to calculate flipped cells in the worker. The worker makes a slice of flipped cells for the section of the world it has calculated, which is sent back to the broker. The broker then combines each slice received by the workers and sends this to the local controller using an RPC call, which is then sent to IO using the events channel. This distributes computation between workers and reduces computation in the broker, yet increases communication overhead, as more information is sent back to the broker by the workers. As shown in Figure 15, this implementation is slightly slower, due to this communication overhead increase.