Filip Hrehovcik (zu24411), Filippos Babalitis (gw24883), Harley Huang (dv24243)

Parallel Implementation

1. Functionality and Design

1.1 Features and Problems Solved

Our first Game of Life implementation was a serial/single-threaded program which iterated over the game board, represented as a 2D slice, and computed the next iteration cell-by-cell. We quickly went on to parallelise this solution and created ‘worker’ goroutines to parallelise this, each worker being responsible for a different strip of the game board. To ensure that the entire image was divided as evenly as possible, non-integer divisions were rounded to the nearest whole number.

The game board wraps around at its edges, meaning cells at one edge of the board are adjacent to those at the other end. Hence, in order to get the adjacent values of a given cell, we needed a way to achieve this with regards to traversing the 2D slice of the board. This was initially solved using modular division and is further discussed during the Benchmarking.

An additional ‘ticker’ goroutine was also created to count up and report the number of alive cells in the current turn every 2 seconds. Furthermore, to enable real-time control of the simulation, key presses are collected and sent via the *keyPresses* channel. This ended up significantly changing the program flow as it meant adding a *select - case* block at the start of each new turn, which would read the *keyPresses* channel for any inputs and act accordingly, otherwise simulating the next turn as normal.

1.2 Goroutines and Synchronisation

The program starts by calling *gol.Run()* and *sdl.Run()* inside *main()*, which each handle the logic for the simulation and the SDL display respectively. These two communicate with each other through the *events* and *keyPresses* channels; *events* is used to update the SDL window with changes to the state, while *keyPresses* are used to pass user inputs to the simulation.

The *distributor()* function is where each iteration of the simulation occurs. It creates each of the worker goroutines, which return their slice of the simulation using channels. *startIo()* is asynchronously called alongside *distributor()*, and is responsible for reading in the initial state and saving states as a PGM. These communicate through the channels in the *ioChannels* struct.

The implementation of the ticker gave way to the possibility of data races as the program could update the slice containing the game board and the turn counter while these were being read by the ticker. To address this, a mutex lock was placed around this critical section to avoid simultaneous reading and writing.

2. Benchmarking and Analysis

2.1 Experimental Design

All of the following parallel implementation benchmarks were run on the same machine featuring an Ubuntu Linux 22.04 environment hosted in WSL and powered by an AMD Ryzen 7 7480S processor, which has 8 cores and 16 threads. Every benchmark performed 1000 iterations of the Game of Life simulation and obtained the average over 6 repetitions to strengthen the reliability of results.

2.2 Use of Multiple Workers

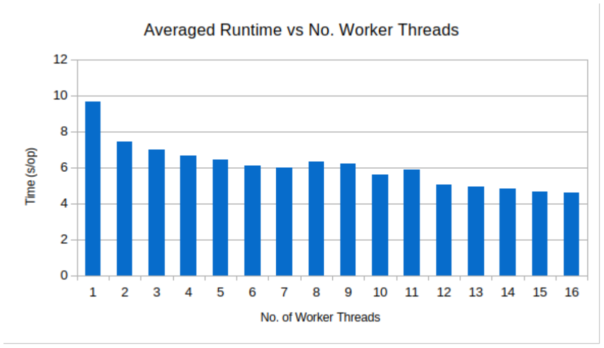
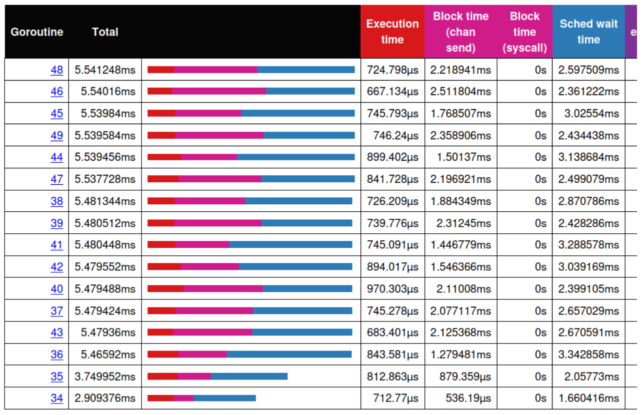


Figure 1: Runtime of parallel implementation against the number of worker threads

Figure 1 shows an overall improvement in the runtime of the algorithm when leveraging more threads, and at 16 workers the simulation runs roughly 2.09 times faster than with only 1 worker. It's also clear that adding more worker threads does not proportionately decrease the runtime and gives way to diminishing returns. This is because while adding more workers allows for more of the simulation to be computed simultaneously, overheads are incurred by creating and destroying goroutines, as well as through scheduling, since goroutines compete for CPU time. Communication using blocking channel calls also mitigates performance gains.

In our implementation, the *distributor()* receives the computed slice from each worker through unbuffered channels, allowing us to clearly exhibit the limitations described above with the following table, shown in Figure 2:



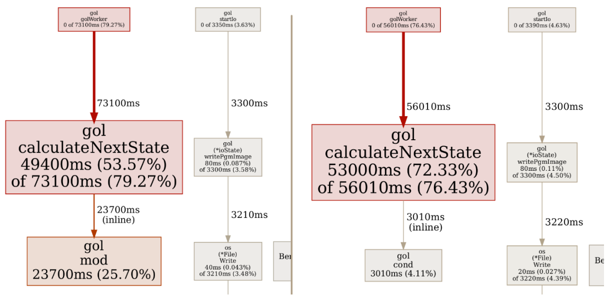
Goroutine	Total	Execution time	Block time (chan send)	Block time (syscall)	Sched wait time
48	5.541248ms	724.798µs	2.218941ms	0s	2.597509ms
46	5.54016ms	667.134µs	2.511804ms	0s	2.361222ms
45	5.53984ms	745.793µs	1.768507ms	0s	3.02554ms
49	5.539584ms	746.24µs	2.358906ms	0s	2.434438ms
44	5.539456ms	899.402µs	1.50137ms	0s	3.138684ms
47	5.537728ms	841.728µs	2.196921ms	0s	2.499079ms
38	5.481344ms	726.209µs	1.884349ms	0s	2.870786ms
39	5.480512ms	739.776µs	2.31245ms	0s	2.428286ms
41	5.480448ms	745.091µs	1.446779ms	0s	3.288578ms
42	5.479552ms	894.017µs	1.546366ms	0s	3.039169ms
40	5.479488ms	970.303µs	2.11008ms	0s	2.399105ms
37	5.479424ms	745.278µs	2.077117ms	0s	2.657029ms
43	5.47936ms	683.401µs	2.125368ms	0s	2.670591ms
36	5.46592ms	843.581µs	1.279481ms	0s	3.342858ms
35	3.749952ms	812.863µs	879.359µs	0s	2.05773ms
34	2.909376ms	712.77µs	536.19µs	0s	1.660416ms

Figure 2: Table analysing the worker goroutines for a program trace, for 16 workers

The table shows that each goroutine spends very little time actually performing calculations, rather, the majority of its time is spent waiting to be executed while scheduled, and then waiting again to send its computation back to *distributor()*. Goroutines 34 and 35, the first of the created workers, spend significantly less time waiting as *distributor()* receives from each worker in the order of their creation.

2.3 Modulus vs. Conditional Statements

As mentioned previously, our initial implementation used modular division when traversing the slice containing the board in order to wrap around to the other edge. Because modular division is relatively computationally expensive and occurs extremely frequently during simulations, we instead replaced it with conditional statements to adjust the indexes to the other edge of the slice when needed. This ended up significantly reducing the runtime, and after isolating each operation into dedicated functions, we were able to generate the CPU profiles in Figure 3 which demonstrate how the associated execution time decreases when using conditional statements.



gol golWorker 0 of 73100ms (79.27%)
gol startIo 0 of 3350ms (3.63%)
73100ms
3300ms
gol calculateNextState 49400ms (53.57%) of 73100ms (79.27%)
gol (*ioState) writePgmImage 80ms (0.087%) of 3300ms (3.58%)
23700ms (inline)
3210ms
gol mod 23700ms (25.70%)
os (*File) Write 40ms (0.043%) of 3210ms (3.48%)
gol golWorker 0 of 56010ms (76.43%)
gol startIo 0 of 3390ms (4.63%)
56010ms
3300ms
gol calculateNextState 53000ms (72.33%) of 56010ms (76.43%)
gol (*ioState) writePgmImage 80ms (0.11%) of 3300ms (4.50%)
3010ms (inline)
3220ms
gol cond 3010ms (4.11%)
os (*File) Write 20ms (0.027%) of 3220ms (4.39%)

Figure 3: CPU Profiles of code using the modulo function (left) and conditional statements (right)

2.4 Extension: Memory Sharing

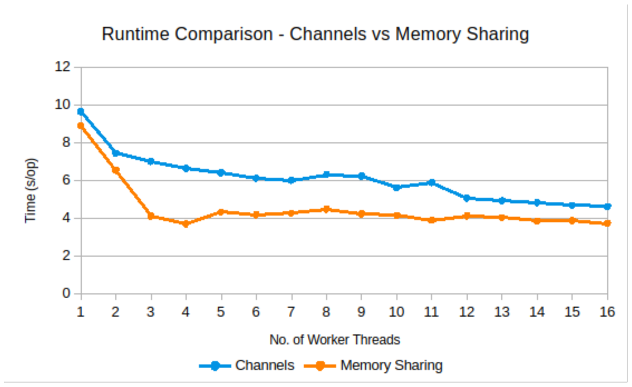


Figure 4: Runtime comparison of the parallel implementation using channels against the memory sharing implementation

The extension was completed by creating structs which emulate the behaviour of buffered and unbuffered channels while still using pure memory sharing. These include mutex locks and conditional variables to enable synchronisation as well as pointers to the data being sent across, and the implementation of a custom interface that describes the blocking as well as the non-blocking interactions of channels. Having done that, we replaced all the channels in our regular implementation with these custom channels, adding wait groups where necessary to prevent data races, and also replaced select statements with equivalent logic.

As shown in Figure 4, this implementation greatly improved on the runtime of the simulation. This was due to completely eliminating the communication overhead normally incurred by channels; in particular, the efficiency of reading in and writing out a PGM file was greatly improved this way, as the memory sharing implementation simply passes a single pointer to the 2D slice as opposed to passing the slice byte-by-byte. However, it is worth noting that the methods used for synchronisation also incur their own delays in exchange - altogether the tradeoffs between these two approaches are worth investigating further.

2.5 Points for Improvement

One possible improvement that could be made would be to reduce the size of the board slice by 'packing' 8 cells into a single byte, as opposed to having just 1 cell per byte. This could greatly reduce the memory used by the world slice and each bit could be accessed using bitwise operations. Challenges would include iterating over the slice as you would have to iterate over each bit, handling situations where the dimensions of the image are not evenly divisible by multiples of 8, and investigating whether performing the necessary bitwise operations would have any performance drawbacks.

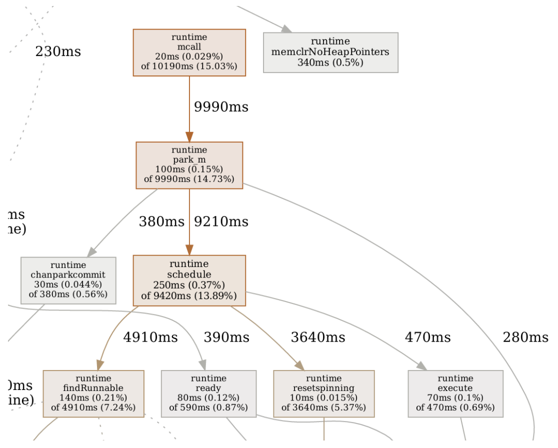

230ms
runtime mcall 20ms (0.029%) of 10190ms (15.03%)
runtime memclrNoHeapPointers 340ms (0.5%)
9990ms
runtime park_m 100ms (0.15%) of 9990ms (14.73%)
380ms
9210ms
runtime chanparkcommit 30ms (0.044%) of 380ms (0.56%)
runtime schedule 250ms (0.37%) of 9420ms (13.89%)
4910ms
390ms
3640ms
470ms
280ms
runtime findRunnable 140ms (0.21%) of 4910ms (7.24%)
runtime ready 80ms (0.12%) of 590ms (0.87%)
runtime resetspinning 10ms (0.015%) of 3640ms (5.37%)
runtime execute 70ms (0.1%) of 470ms (0.69%)

Figure 5: CPU Profile showing runtime functions associated with scheduling

Another improvement would be to reuse worker goroutines to avoid re-creating new ones each iteration of the simulation. As shown in Figure 5, a significant portion of execution time is spent inside functions associated with goroutine scheduling, meaning reusing goroutines for larger tasks could be an effective way of improving runtime.

Distributed Implementation

1. System Designs

1.1 Single Worker

At the first stage of the distributed implementation, a single AWS node (worker) is used to handle all Game of Life logic. The worker runs an RPC server that registers its core methods and listens on a specific port. When a local controller dials this port, an RPC session is created allowing the client to call the server's methods. The current state of the game board and the turn variable are shared across RPC methods as global variables. These are protected by a mutex to prevent data races.

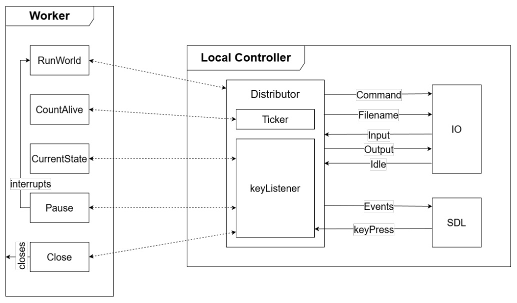


Figure 6: UML diagram featuring only a single worker

The local controller is made up of three components as shown on Figure 6. Each component represents a goroutine, and the arrows correspond to their channel communications. *Distributor* starts two additional goroutines, *ticker* and *keyListener,* immediately before the RPC request is made and terminates them as soon as the RPC call finishes to ensure correctness. The *ticker* itself does a RPC call every two seconds to obtain the current alive cell count. The *keyListener* listens for keyboard input arriving from the *SDL* component through the *keyPresses* channel and handles the keys ‘s’, ‘q’, ‘k’, and ‘p’ performing defined functionality. When the simulation is paused, *keyPresses* are routed to a dedicated loop inside the distributor, which waits for input and determines whether to remain paused, resume, produce an output, or terminate. The *keyListener* enters normal operation again as soon as it receives a resume trigger.

1.2 Publisher-Subscriber with a Broker

For this stage, a number of AWS nodes cooperatively compute the next state of the game board. The system depicted in Figure 7, features a broker to decouple the local controller (publisher) from the worker nodes (subscribers), making it easier to test with different numbers of workers. The broker is started first and begins listening on a port. Afterwards, the individual worker nodes are initialised, subscribing to the broker themselves. Finally, the local controller invokes *BreakWorld* to publish the task to the broker.

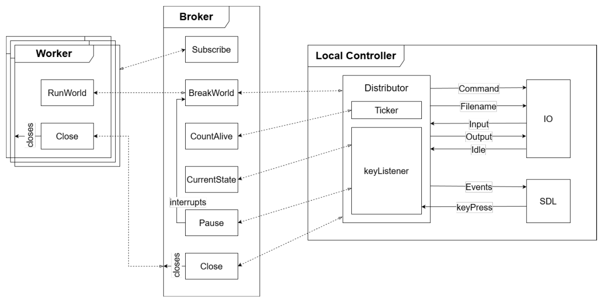


Figure 7: UML diagram of a pub-sub with a broker

The broker splits the board by assigning each worker a specific range of rows, including two halo rows. On every turn, it calls *RunWorld RPC calls* to all workers and updates the internal game board state according to their responses. When the controller calls *Close,* the broker forwards the request to every worker before closing its listener. This causes all workers to close their listeners and terminate.

1.3 SDL Live View

In this stage, SDL support is added to the previous broker pub-sub system. The distributor launches a new goroutine, *sdlSender*, which repeatedly issues RPC calls to the broker to obtain flipped cells on each turn. The response is forwarded to the SDL component over the *Events* channel. When the simulation is paused or stopped, the broker returns an RPC response with *Success* = *false*, signaling *sdlSender* to stop requesting further updates. The interactions between components are demonstrated in Figure 8.

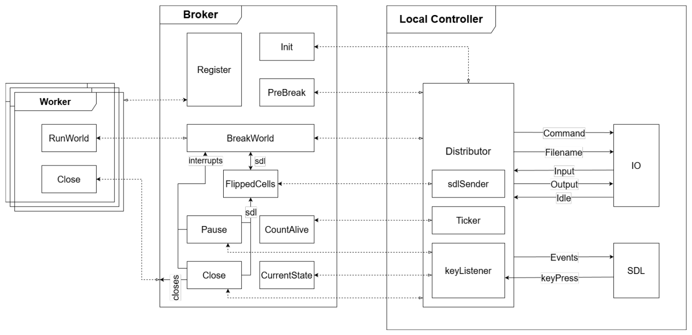


Figure 8: UML diagram of a system implementing SDL

1.4 Multiple Clients

This custom extension features support for handling multiple clients (local controllers) simultaneously, using the broker-based pub-sub architecture. The broker assigns each client a unique index and stores the task state of all connected clients. It splits the total workload evenly between workers. When the ‘k’ key is pressed, only the associated client state will be removed, rather than shutting down the broker and all workers as in pub-sub. The overall design is shown in Figure 9.

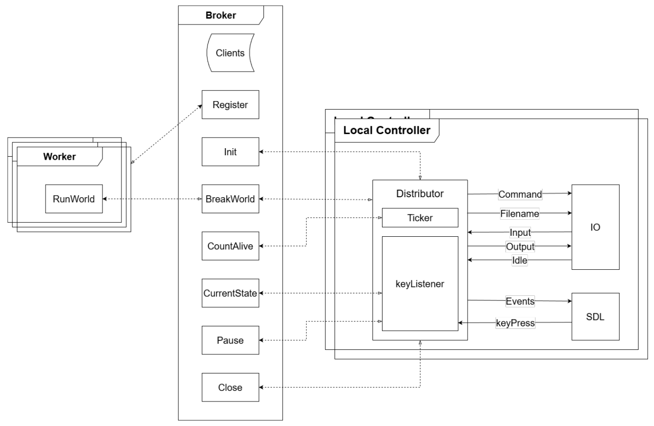


Figure 9: UML diagram of a multi-client broker system

1.5 Fault Tolerance

This extension implements a fault tolerance mechanism for servers. The architecture is illustrated in Figure Z. Each server contains the full broker logic and can take over as the broker if the need arises. When a server starts, if the address flag for the broker differs from its own address, it calls the broker to obtain the list of all the other server addresses, subscribing to both the broker and these servers. The local controller also stores addresses of every server registered with the broker. If a worker node goes down while running a task, its workload is reassigned to a free worker. If no workers are free, the task will be redistributed to all the running workers. If all workers fail, the broker will call its own worker methods so that the simulation keeps going. The *ticker* now periodically requests the current board state and stores it as a backup in case that the broker node goes down. If the broker fails but some workers are still running, the local controller will connect to the first running worker in its address list and make it a new broker as shown in Figure 10:

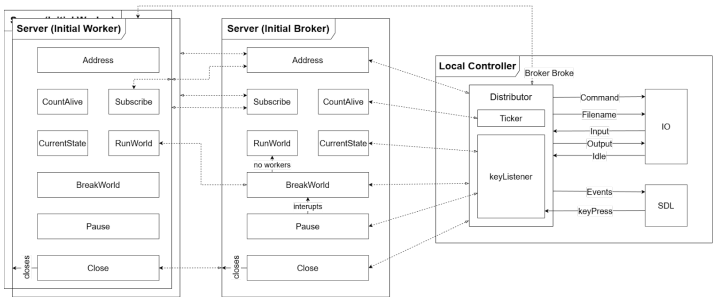


Figure 10: UML diagram of the fault tolerance extension

1.6 Halo Exchange

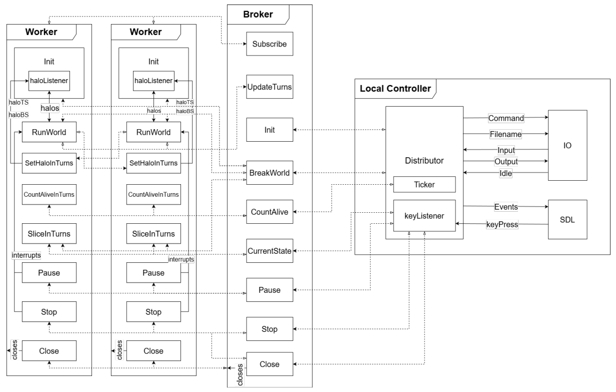


Figure 11: UML diagram of halo exchange extension

Figure 11 depicts the program structure of the halo exchange between workers which reduces communication overheads. Instead of resending the whole world from the broker every turn, each worker updates the halo of its two neighbors. Also, if there is only a single worker, it will just update its halo without using RPC calls. The haloListener waits for top and bottom halo updates before unblock RunWorld to calculate the next state. The workers also store all the previous states down to the overall minimum turns completed. This is achieved by calling the broker every turn to update its completed turns and get the lower bound. These stored states are required by the broker to get the “current state” when handling CountAlive and CurrentState. When the execution is paused, the workers are blocked and don’t finish the current work in this implementation. This is to prevent workers’ progress after the overall completed turns are lost and affect the efficiency.

2. Benchmark Testing

2.1 Single Node vs. Pub-Sub Broker

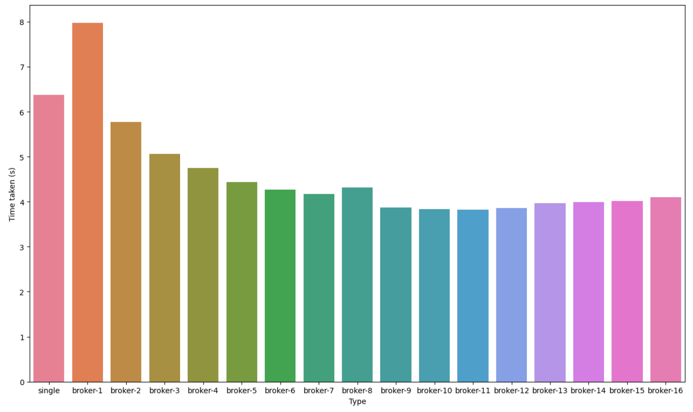


Figure 12: Impact of worker count on total execution time for 1000 turns

Figure 12 puts into comparison the time taken to execute 1000 turns for different numbers of workers. A key comparison is between using a broker with just one worker and a single worker on its own. A broker with one worker essentially adds an extra middle layer which strictly increases communication overhead. Generally, as the number of workers increases, the time first decreases to a threshold, with the most effective configuration being a broker with 11 workers and then starts to increase again. This is because although each individual worker is given a smaller slice, each worker added will add 2 halo rows to the total communication cost, and the constant increase in overhead eventually becomes greater than the diminishing returns of adding a new worker.

2.2 With vs. Without SDL

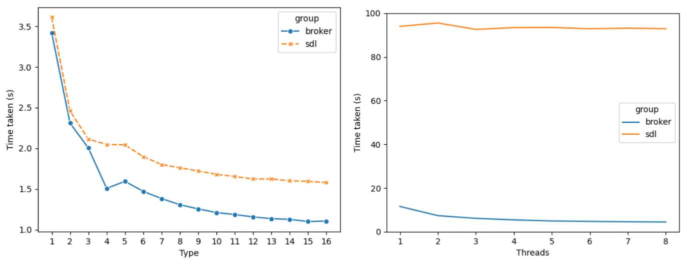


Figure 13 and 14: Comparison of execution time with and without SDL under local and AWS-hosted configurations

The figure 13 above shows the difference between the traditional broker-base pub-sub system and the implementation of SDL, on a university lab machine. The IO time is not taken into account. The latter slows down the execution speed noticeably, but it can be seen that as the number of workers increases, the time difference between a broker with SDL and without when SDL approaches constant. This is due to the fact that the time taken to communicate flipped cells is constant no matter the number of workers.

When the servers are hosted on AWS EC2 Instances, illustrated in Figure 14, the time difference becomes much larger, since the SDL implementation requires making RPC calls from local to AWS node on every turn, being significantly slower than purely local calls.

2.3 Broker vs. Halo Exchange

Figure 15 illustrates the results of benchmarking 100 turns for a 5120x5120 world image using local nodes. The time difference between the two implementations has a constant trend. Initial memory cost is the same for both of them, but halo exchange is more effective as it avoids repeatedly sending the full slices of the board to the workers on every turn.

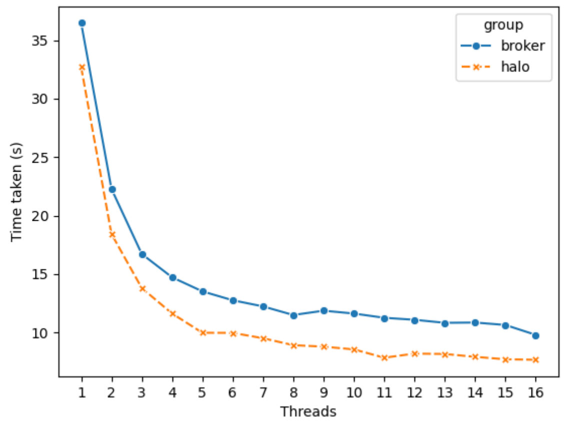


Figure 15: Benchmark results for 5120×5120 board on 100 turns under local execution

Comparison of the halo and normal broker implementation on EC2 instances for 1000 turns on 512x512 image is shown in Figure 16. The broker implementation for a single worker takes the longest as described in 2.1. The time taken for a single worker halo exchange is similar to a single node since it's updating its own halos. The time difference when worker numbers increase is less than the 5120x5120 benchmark, since the world size is 100x smaller, thus repeatedly sending the whole board is less taxing.

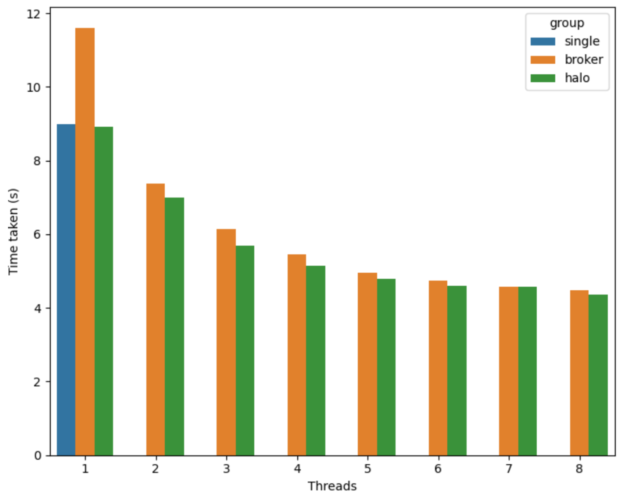


Figure 16: Comparison of halo-based and normal broker implementations on EC2 instances for 1000 turns on a 512×512 board