Simulation of Conway's game of life report

By Sean Xu (tk22444) and Ruoxin Chen (pw22232)

Stage 1 - Parallel Implementation

Part 1 Functionality and Design

1.1 Functionality Implemented

Our parallel implementation is based on lab1's Game of Life logic.
In our original Game of Life logic, it receives a 2D array as a whole board and calculates the next turn. For parallelizing it, we change it to only calculate the next turn for part of the board base on the give start and end index on y-axis.

In our implementation, distributor will divide the board's lines evenly into different parts and send the starting and ending position of the y-axis of each part and the size of this part to the worker. Then, that worker will calculate next state of given part. Each worker will receive a channel for them. The distributor will use those channels to collect the result form workers when all workers finish calculation.

After completing the parallel GOL logic, we use another to implement the function of reporting the current number of alive cells every two seconds and processing SDL keyboard input. We support three cases for keyboard input: "s" for generating a PGM image containing the current state of the board, "p" for pausing and resuming the program and "q" for terminating the program. We also support real time display of the game state in SDL. After calculating all turns, we output final image of the board.

1.2 Problems Solved

The first problem we met was how to pass different workers the data they need. We found that we will not modify the board during the process of calculating next state, so we turned the whole world into an immutable object and passed it to every worker to make sure the board's content is read-only for workers. Workers can visit the data of original board directly. From testing result we find that it would not produce any race because we only write new data to the board after all workers complete their job.

Another problem we met is how to distribute equal amounts of work to workers to decrease the time when waiting all workers finish their work. We first came up with an idea that uses the board's height divided by the number of threads to get the average number so the board can be partitioned into same number of rows. The rest parts will be allocated to last worker. However, we think this operation will cause the last worker to spend too much time dealing with its work if the height of the board is not divisible. Then, we found the reminder always smaller than the number of workers, so we allocate one more row to the workers that be created earlier according to the remainder after the division. This decrease the gap of time between each worker to finish their work when the image's height cannot be divisible by the threads amount. By the way, it also protects the program form shutting down when there are more threads than the height of image.

Part 2 Testing and Critical Analysis

2.1 First version

Figure 1 is the benchmark result of our first version in different threads number. The result shows that the speed of running GOL increase when there are multiple threads. We find that diminishing marginal return appears very quick in our parallel implementation with the image of 512 * 512 and turns of one thousand. When the number of threads is greater than ten the required time even increases. which may relate to the increase of time when creating multiple 2D array and channel communications by the workers.

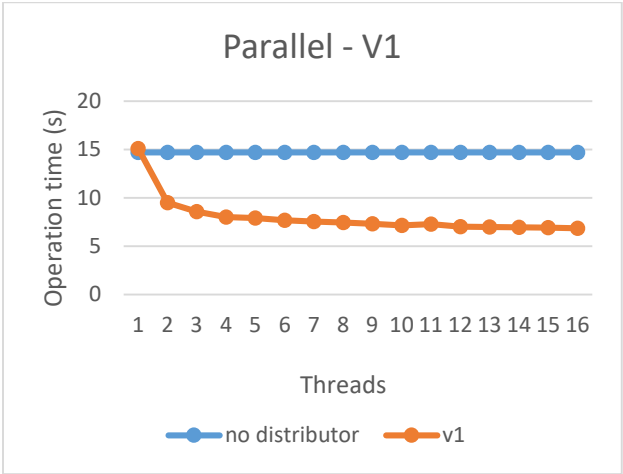

Figure 1: The benchmark result for our first release (on 512x512x1000)

2.2 creation of 2D array Flipped events.

In first version, our worker will create and return a new board slice of the part it is responsible for when the calculation is completed. And for displaying the board in SDL, the worker will pass a Cell Flipped event every time when it finds a cell that needs to be modified. This requires lots of parameters to pass when creating a worker. Also, the number of cells which need to be modified in each worker is different. The blocking of the event channel will cause a large gap between each worker's working time. To solve these problems, we modify the worker so that it only returns the coordinates of all modified cells. These modifications will directly apply to the current board and passed to the event channel after all workers finish their job.
Therefore, the difference between all workers' working time gap is reduced, workers and distributer do not need to build new 2D arrays to store modified boards data. This obviously improves performance, as shown in the figure below.

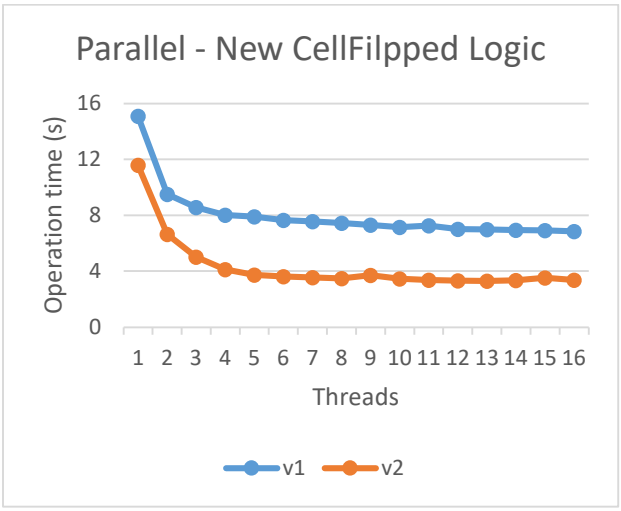

Figure 2: The benchmark result for our second release (on 512x512x1000)

2.3 Merge of count alive cells and keypress

In the original implementation, we set two goroutines for reporting alive cells every two seconds and the keypress for controlling SDL. This causes the ticker to still report the current number of cells every two seconds after the SDL was paused. We suppose it's not necessary, but if we make the changes in ticker and main function consistent, when main function pauses ticker also pauses, for example. To solve this problem, we merged the two goroutines and used select to determine whether the ticker and keypress channels were triggered. This successfully solved the problem and reduced the use of one goroutine.

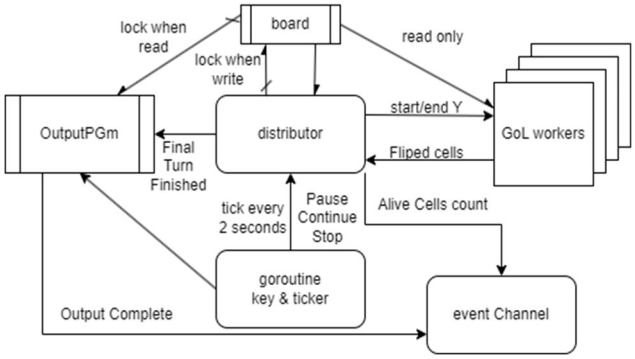

Figure 3: The structure of our final version of parallel implementation

2.4 Possible Improvement

Our program must create workers and collect data from them in every turn which brings additional cost especially when the number of turns is very large. A possible solution is let each worker exchange the changes of halo region after each turn and directly proceed next turn calculation.

We did not find a good way to change our implementation to use memory sharing only instead of using channels yet. I believe we can do that if we have more time.

Stage 2 - Distributed Implementation

Part 1 Functionality and Design

1.1 Functionality Implemented

Our distributed implementation is based on parallel part. We use RPC to make a publish-subscribe model to allow us run methods remotely. Besides, we added a shared "stubs" file to store all data structure, therefore all go files can use the same structure to call methods which avoid unexcepted result when changing the function of any of them.

We add a broker to reduce the coupling between controller and server. As well as enabling multiple servers to process GOL simultaneously. In our program, we use the same splitting logic as the parallel version to allocate an average task to each server. Broker will split the board and call the methods in server to run the game after receiving the whole board sent by controller. To reduce communication overhead, our broker will only send slice of board needed to be processed and adjacent server addresses to each server. Servers will get the required halo region from their neighbouring server. Broker will collect the list of cells whose state should be changed from server, then apply it to board and send the board back to controller.

We added a ticker that calls broker's method to get the alive cells count every two seconds

after we finished GOL operations in distribution part. We also supported keyboard control. When the user press “s”, controller will call the GetWorld method in broker to get current turn and output PGM image. When user press “p”, controller pause/resume broker from dealing with the next turn and return the current turn number. When user presses “q”, our controller will output PGM image after acquiring the current board from broker, then exit. This process will not exit the broker and server. Broker and server will reset automatically and run a new board when a new controller connects. When user presses “k”, controller, broker, and all servers will be close.

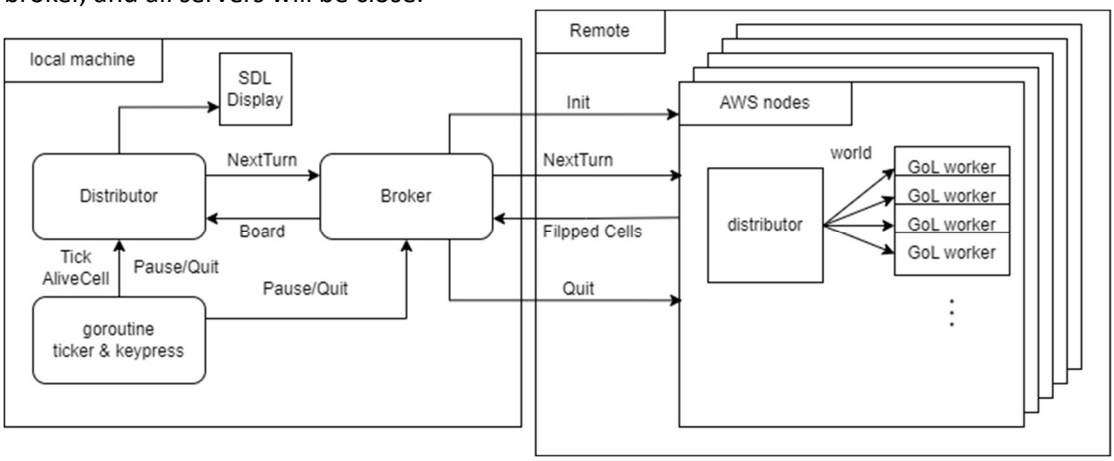

Figure 4: The structure of our submitted version of distributed implementations

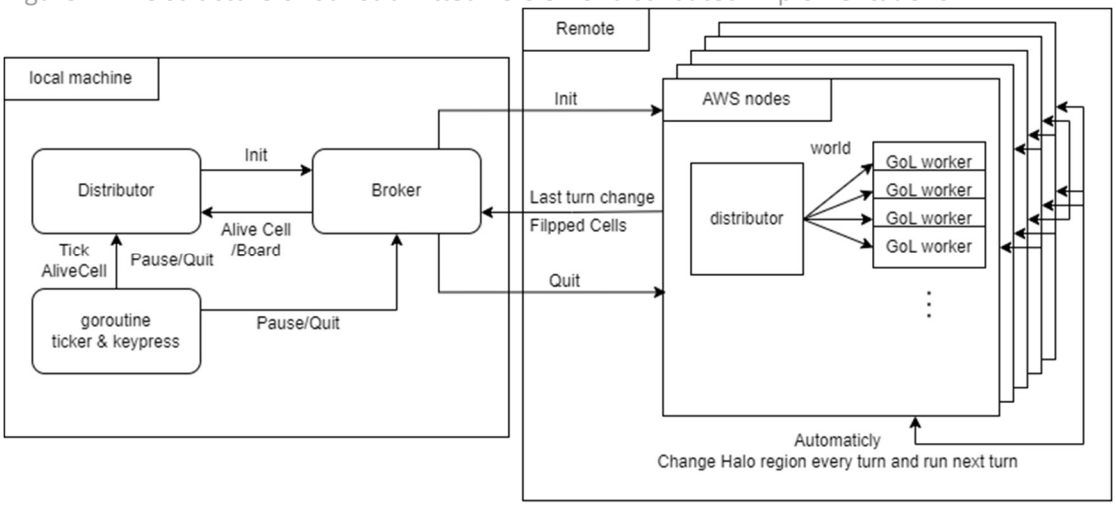

Figure 5: The structure of another version of us while server will automatically run every turn.

1.2 Different ideas and implementation

We managed to write two implementations in distributed part. Although they both use Halo exchange, the exact way they run the game is quite different.

In the first version (the version we submit), our distributor components will synchronize in each turn. It supports the real-time SDL display for current game status. Our broker will initialize the server first, passing part of board which needs to be processed and the address for halo exchange. At this point, the servers won’t start running. Controller will send NextTurn requests in every turn to broker after initialization complete. Then, broker will send same requests to all servers. Server will first do halo exchange to get halo region then return the next turn’s results back to broker after complete calculation. Broker will send them back to controller after collect and apply results to the board. However, we discovered that this

causes a large amount of communication overhead.

Consequently, we had another try. In this version, our server starts to run after receiving broker's initialization request. Server get halo region from neighbour server in a separate goroutine and wait for other servers to accept their halo region at the beginning of every turn. Broker will send a request to server only when controller needs to output images or calculate the number of alive cells. However, this method will cause one turn gap between servers at some time due to the time every server finish one turn is different. To solve this, we store the changes made in the last turns in a separate list, and the broker automatically rolls back the changes made if server has a gap of turn when it receives the data back from the server. This has been tested to significantly reduce communication overhead and can make runs nearly 14 times more efficient on the AWS environment. Nevertheless, we were unable to implement SDL display in this release.

Part 2 Testing and Critical Analysis

2.1 Test conditions

To quantify the performance of different versions of our distributed implementation all our benchmarks were run on 4 AWS t2. micro servers in the same network environment.

2.2 First version

In our first version, we let the broker send the data needed for each server including two extra lines in every turn. After remote worker finish their work, we collect the results returned by the server. We found that this is too inefficiently. We then started to work on the halo exchange. We first make halo exchange in the simplest way (synchronization every turn), we also modified the brokers and controllers to achieve real-time presentation of the game state in the SDL. However, in our tests, using the halo exchange in a turn-by-turn synchronized did not significantly improve efficiency. the addition of SDL support even slowed it down. In addition, we found that in both implementations, as the number of nodes in AWS increased, the operation became less efficient instead. We conjecture that this is because the increased remote communication overhead of multiple servers far outweighs the reduced time spent processing images by multiple AWS nodes.

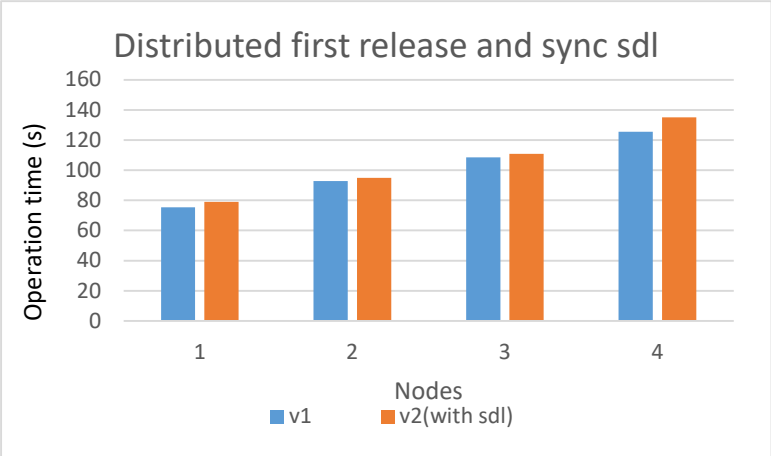

Figure 6: The benchmark result for our first and second (with halo and SDL) version (on 512x512x1000)

2.3 run GOL by Halo exchange without broker

Synchronizing servers and brokers every turn brings too much communication overhead. To

solve this problem, we rewrite the method calls between brokers and servers. After the broker sends the required data to all servers, all servers will automatically run each round and exchange halos with each other without communicating with the broker, which greatly reduces the communication overhead and makes the game much more efficient especially when running on multiple servers at the same time.

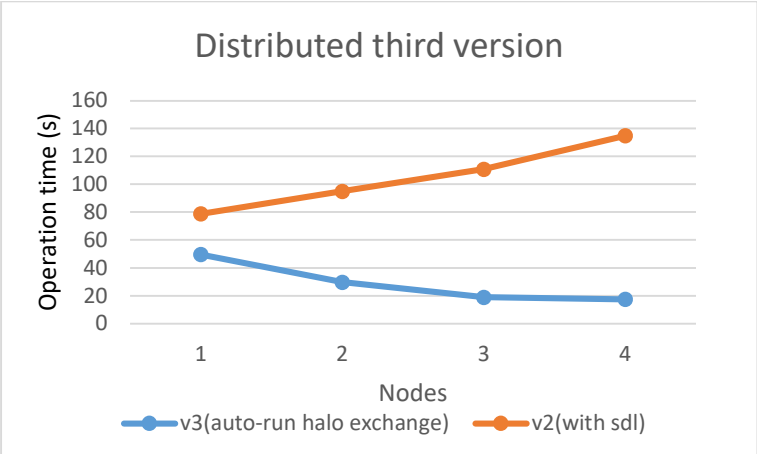

Figure 7: The benchmark result for our third version (auto-run with halo exchange) (on 512x512x1000)

2.4 distributor in remote worker

After completing two versions of the distributed implementation, we added the parallel component to Server as well. After testing, we found a slight improvement in efficiency with multiple threads. However, it stopped improving beyond 4 threads. It even requires longer time in benchmark beyond 12 threads. We suspect that this is related to our AWS server has few numbers of vCPUs and the size of the images is also quite small.

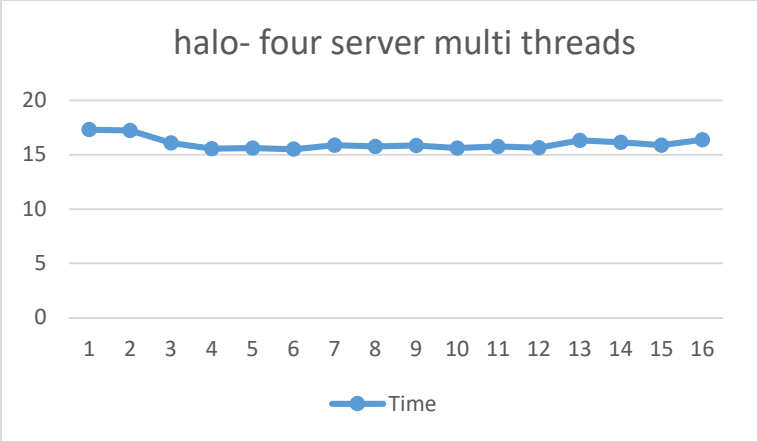

Figure 8: The benchmark result for multi-threads in our third version of distributed (on 512x512x1000)

2.5 Possible Improvement

While the servers are running, if one server unexpectedly shuts down, our entire distributed program shuts down. A possible solution would be for the broker to simply reset all servers when it finds one disconnected, and then reassign each server's work and the server address they use for halo switching based on the number of servers remaining.

In our autorun halo swap, there will inevitably be channel deadlocks if any servers are accidentally shut down, so we didn't use this version as the one we submitted to blackboard, even though it's very efficient.