Ally Park (yg21152), Hyunho Kim (aq20307)

1. Parallel implementation

Functionality and Design

In General, distributor make evolution on given world allocating computation between multiple workers for restricted number of turns. Distributor interacts with other goroutines in computation progress using channels. In detail, interacting IO goroutine, it creates a size-specified 2D slice to store the image getting in afterwards byte by byte. After the image input is done, the distributor runs two function literals on goroutine simultaneously for interaction with SDL goroutine. One is for reporting computing turn and the number of alive cells by every 2 seconds. This information is displayed on the terminal. The other is for capturing keypress. This enables for users to be able to interact with SDL live progress display while program is running. According to the instruction on the terminal, users can pause, resume, quit the program and generate the PGM image of current turn. When the evolution reaches a limited number of turns, the distributor notifies SDL that all calculations are finished, communicates with the IO to create a PGM image, and terminates the program.

In world evolving logic in game of life, one function calculates the number of neighbouring alive cells including cell itself in given state. Following the rules of game of life, subtract the number of alive neighbours by 1 if the cell is alive in given world. This calculation determines whether the cell lives or dies in the next world. This cell-by-cell calculation process repeats for each cell in the world. When the number of threads is not one, the y-axis of the world is divided by the number of threads. Each worker is responsible for the part of the world sliced. These results are merged for the world for the next turn.

There are two version of parallel implementation. One passes the results of parallel worker calculation through the channel. Another implementation uses memory sharing. Without using channel, the calculation results are stored in shared memory using mutex lock to shelter a shared source from concurrent access by multiple processes [1].

Problem & resolve

Although the computation mechanism in game of life was successfully implemented, it was not able to pass the test and visualise the evaluation of the world. This problem occurred because the interaction structure between the distributor and other goroutines was not fully used. Figuring out when the appropriate values are needed to pass, implementation passed relevant values in specific situation through events. This enabled passing test and visualising.

Benchmark Test & Critical Analysis

In this stage, the first paragraph will describe the settings for the experiment, what variables were used, the execution procedures, and the hardware what used in this experiment. Furthermore, the methods also will be explained that used to achieve results with less error and higher accuracy ; with various values entering the params(p.turn, p.imageheight…). The results of the tests consisting of this methodology will be shown. In the second paragraph, after that, the overall trends and improvements will be discussed. The benchmark results of the implemented memory sharing will analysed by comparing it with the non-memory sharing version. Finally, the paragraph will Analyse the advantage of the implementations, and critically address the limitations and problems of the tests carried out. Furthermore, from point of view of hardware, the obtained results and uncertain consequences components would be explained.

To obtain more comparable, accurate data, with various params, a number of benchmarks were taken by a Linux lab machine that was connected by X2go client. In this environment, since the lab machine is connected by X2go client, SDL was couldn't activated. In order to handle that problem, all tests are executed with noVis flag, which disables the SDL window, preventing any visuals from being shown while running tests. To get more accurate results, all benchmark tests are conducted five times in the same environment, and the number of turns was given as 1000 to get higher accuracy.

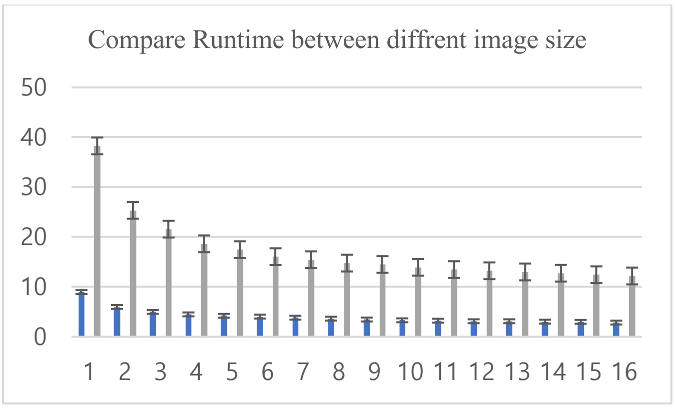

Figure 1. A graph showing the change in runtime time(second) according to the image size and the number of workers (grey: 512 x 512, blue: 256 x 256).

The above graphs represent the runtime according to the number of workers, and each image size. In figure 1, as the number of workers available increases, meaningful results that effectively decrease the runtime of the distinct size of the images were identified. There was a characteristic aspect of the graphs: as the number of workers available increases, runtime decreases rapidly in the beginning, and then at some point, the decline is slowing down. Regarding the unpredicted phenomena, based on our research, it is estimated that as the number of workers increases, absolute amount of decrease in computational range decreases; subsecuently the trend of decreasing runtime as the number of workers increases is mitigated as the number of workers increases.

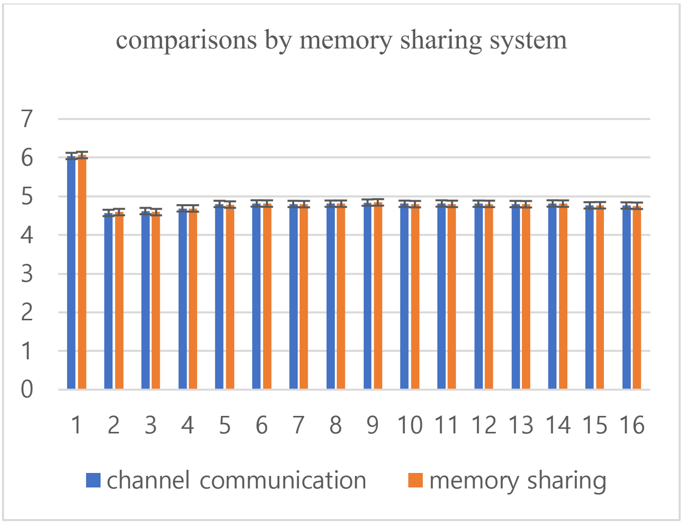

Figure 2. runtime difference between channel communication implementation and memory sharing implementation (image size: 512x512, turn: 1000)

After completing the memory sharing distributed system (Mutex) extension, there was a problem with benchmark testing via x2goclient. Hence this test process was performed on a personal PC (mac, 8 cores). Unlikely the previous test which was conducted according to the size of the image and the number of available workers using channel communication, however, this test was conducted to compare the difference between the implemented version using memory sharing (mutex lock) and the method using channel communication. There are two points. First, looking at both the 5 12 x512 graph in Figure 1 and the cHannel communication graph in Figure 2, the two tests were performed using the same code and parameters, but they show completely different trends. These differences show the relationship between differences in the benchmark test environment and trends in benchmark test results. The next point is the runtime difference in how the worker passes the value in Figure 2. The most remarkable point was observed when the results of the memory sharing were derived; as the test was repeated to obtain the average value, the figure became close corresponding to the non-memory sharing version.

Conclusion

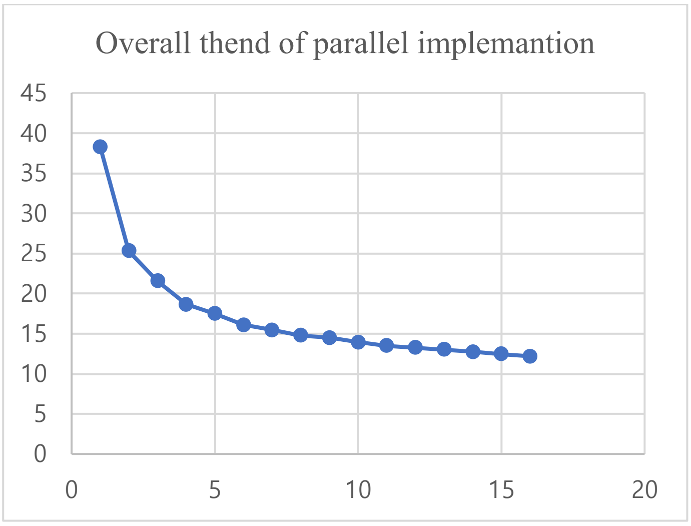

Figure 3. The aspect of runtime which corresponds to the number of workers (512 x 512 image size)

As the graph shows, the runtime reduced as the number of workers engaged. A runtime that reduces by half as the number of workers doubles would be the ideal improvement [2]. However, the observed speed was 3.49 times faster with 16 workers compared to the only one involved with a single worker. Regarding this point, it could be estimated that there were some causes with this acquired result. During the test calculating the runtime from our implementation, not only the time it takes for our game of life logic implementation using multiple workers but also, not paralleled processes are taking more time:

such as receiving images from Io goroutine, sending information between goroutines via channels, etc... Those steps make more difficult to derive precise results. Also, as mentioned in the paragraph above, at some point in the derived graph, a reduction of the decreasing rate of runtime were recognized; which can be addressed from the hardware perspective used in this test. A cpu basically can handle only one thread at a time. Analyse from this perspective, the lab machine has 6-cores, and the cause is regarded to be derived from the issue; the implementation and benchmark test require more thread operations than the number of physical cores in the lab machine. It enables more than one threads to execute on a single core (lab machines are allocated two threads on a core); more threads mean more work can be done in parallel. Hence, seems hard to derive get a result close to the theory result from this environment.

2. Distributed Implementation

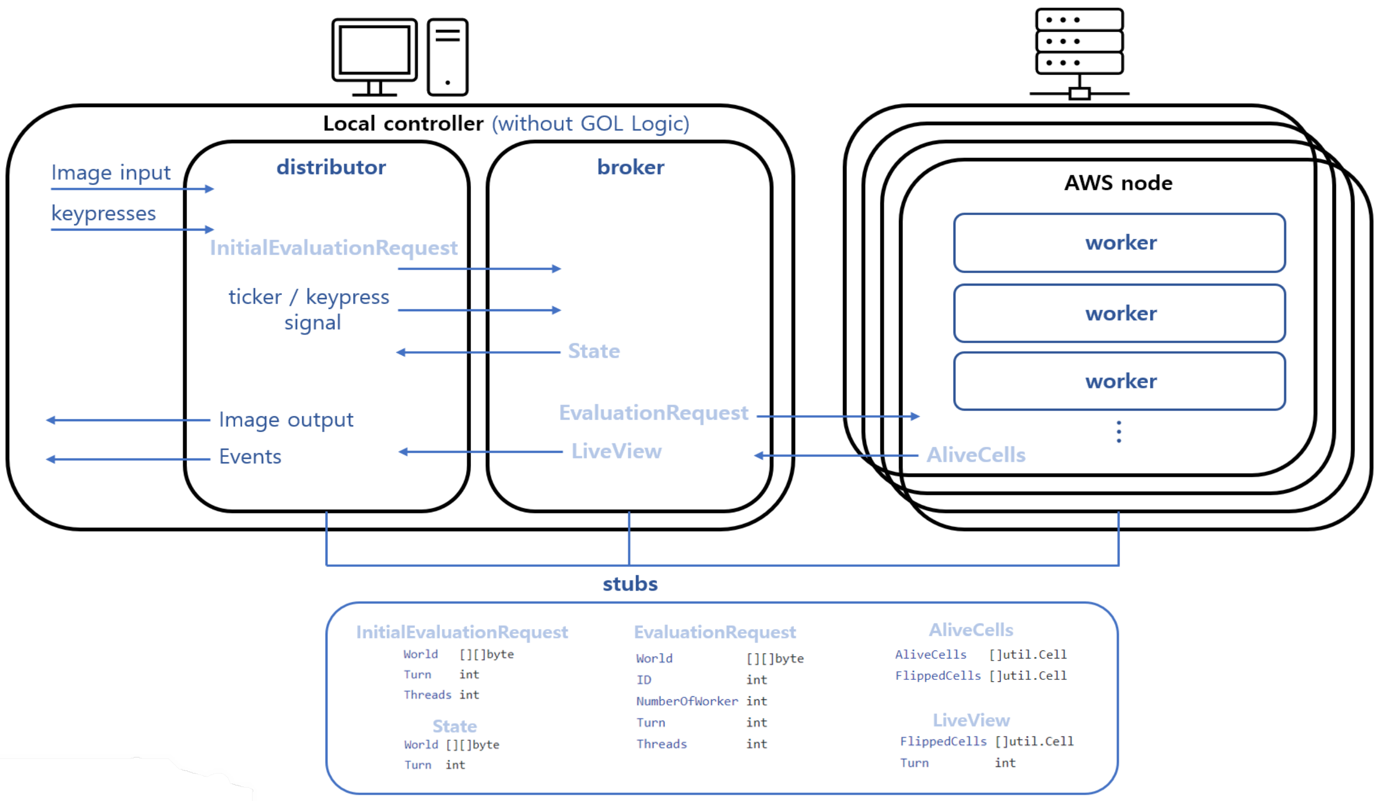

Figure 4. Diagram of distributed system design

Functionality and Design

In distributed implementation, the logic of game of life mechanism is moved into AWS nodes. Depending on the number of open servers, the broker can connect to the server through using flag. When multiple AWS nodes addresses are entered from the command line, even if there is a server that fails to connect, this implementation can execute the task successfully if at least one server and broker are connected. The connected servers are stored in a slice on the broker. Each server calculating a part of the world sliced perpendicular to y-axis.

World evolving computations in AWS nodes are called from distributor via broker. The distributor uses RPC call to the in broker as initial irritation of the program. This call sends initial world to be evolved with the number of turns to evolve and the number of threads. As this implementation is scalable not only for the number of AWS nodes, but also for the thread count. The thread count is used in every AWS node to compute the next state in parallel (slicing part of world perpendicular to x-axis) using multiple workers. Once the computation is done for all given turns, distributor get computed world state as a response of RPC call.

For the RPC call, Client.Go method is used instead of Client.Call method to interact with calculating world state. Using two-way RPC connection between distributor and broker, following functionalities are enabled.

1. displaying the computing turn and the number of cells on the terminal: ticker uses RPC call for every two second to get computing world state
2. displaying live computation progress view: broker notifies that turn is completed and gives information about flipped cells
3. interacting with program using keypress: distributor pass the keypress to broker and get computing world state

Problem & resolve

The first difficulty in the distribution part was implementing communication between the distributor and the server. It wasn't difficult to remove the logic from the distributor, paste it on the server, and let the distributor uses the logic on the server. However, it was quite unclear where to put the RPC connecting code on the distributor as it was not in the main package. Reviewing the starting point of the program, the fact that when the program is executed, a function "Run ()" is called from the main and this function calls the distributor is found. Based on newly recognized information, the code for connection with the server was inserted at the starting point of the distributor function in "distributor. go". The connection with the server worked properly.

When the communication between the server and distributor was implemented the first time, the RPC call was made from the distributor to the server for each turn in the distributor to evaluate the given turn. However, when the code was implemented in this way, there was no distinct difference from the parallel implementation. In order to fix this problem, we found that we had to move the for loop into the server, not the distributor, which to evolve the given number of turns. There was another following crucial problem, how to get the state of the middle process of computing as needed, such as for ticker or key press.

Although using Client.Go method rather than Client.Call, the first attempt was to store the calculation process in response pointers on every turn was fail. Unfortunately, the distributor could not access the values stored in the pointers until the method returned. The second attempt was to use a two-way RPC connection. Using this scheme, we created a ticker through goroutine inside of the computation loop of the server and sent the intermediate calculation process to the distributor every 2 seconds through the client call. The ticker worked well, but this method still could not access the loop under calculation and disturb it, so it could not successfully implement the keypress. The last method used was using a global channel. A function literal is inserted in the computation loop and called using goroutine so that it continuously captures what send via channels. When a key press or ticker needs a state under calculation, the distributor calls the server's method. This method passes the relevant signal to the global channel in the server, and then when the computation loop captures a signal, it sends the computing state to called method. In this progress, the distributor can get a state response from the method.

After that, the broker was created; disconnect the distributor from the server, then link both distributor and server to the broker. However, an error occurred in the process of the broker receiving the evaluated world through the server in each iteration. Using the given test for checking whether the computation logic works correctly, it was confirmed that the test is passed for the 0th and 1st turn but not for the 100th turn. To check the error in more detail, generating the expected world image for all given turns
from parallel implementation, modified the given test to perform the test across all turns. Using the modified test, we were able to see in detail how the calculation process progressed with each turn. We did some debugging to find out which turns, which functions, and under what conditions the calculations went wrong, and found that the calculations worked well without errors. However, when the calculated result was passed from the server to the broker, the value was suddenly changed. The calculated coordinate was (12, 0), and as 0 is regarded as no additional value to pass to, the pointer used the previously calculated y-coordinate. Based on this observation, the code was rebuilt to assign a new pointer to each turn. By allocating a new pointer every turn, the fatal error was resolved.

Benchmark Test & Critical Analysis

In this stage, the environment for benchmark testing for the distributed part; the methodology of building benchmark test, and the reason will briefly explained. In the next paragraph, we will present and analyze the results of the tests conducted with the methodology mentioned above. Finally, we will critically analyze our implementation and the methodology of the tests, then discuss the limitations of the methodolgy , and advantages of our implementation.

For the test of the distributed stage, c4.xlarge instances were used, not the t2.micro which has been used for testing our implementation; the reason is that it has high network performance (bandwidth of 750 Mbps), compared to t.2micro or another instance. We considered that higher accuracy results could be obtained from this higher network performance instance in this stage.

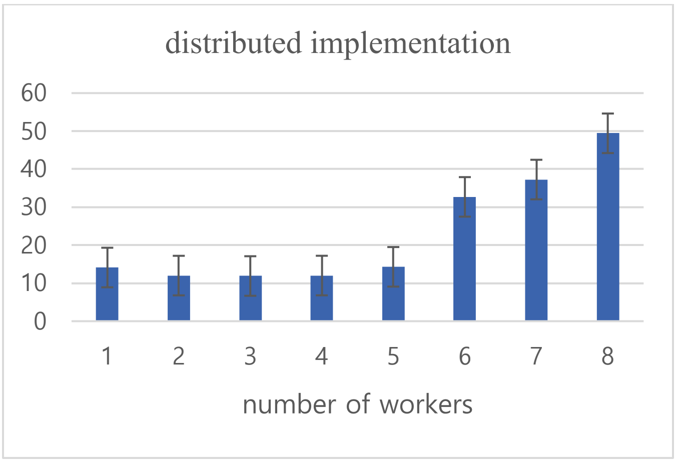

Figure 5. trends in performance with the number of nodes

In the distributed part test, unlike the parallel part, which analyzed the differences in runtime by input various values in one code, tests on this distributed step will analyze runtime differences of various versions implemented with fixed inputs. In the first test, by increasing the number of AWS nodes used from 1 to 8, various tests were conducted to watch how the performance changes accordingly.In the graph above, the graph initially showed a decreasing trend, but at some point, it started to show an increasing trend. According to our research, it is assumed that as the number of nodes increases, the corresponding overhead time increases, subsequently from some point, the total runtime will be increased due to more nodes increases, rather than the time benefit of dividing the computation among nodes and decreasing.

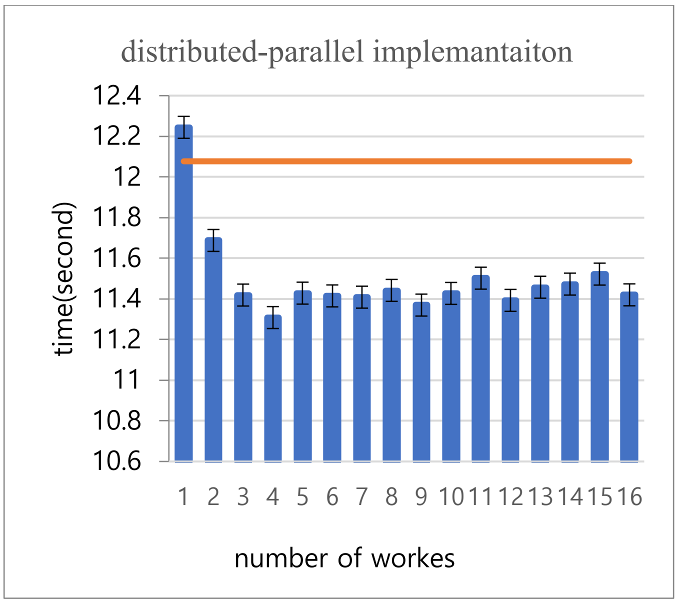

Figure 6. runtime improvement in distributed-parallel implementation and average runtime of distributed implementation.(512x512, turn : 100, number of nodes : 4)

Figure 6 shows the visualized graph based on the results of the benchmark test that was executed using four nodes in the In Parallel Distributed System; one of the extension tasks, which has been implemented (set to 100 turns in a 512x512 image and repeated 5 times in total). This makes comparisons with the average runtime obtained when using four AWS nodes in a non-parallel distributed version visually.

Conclusion

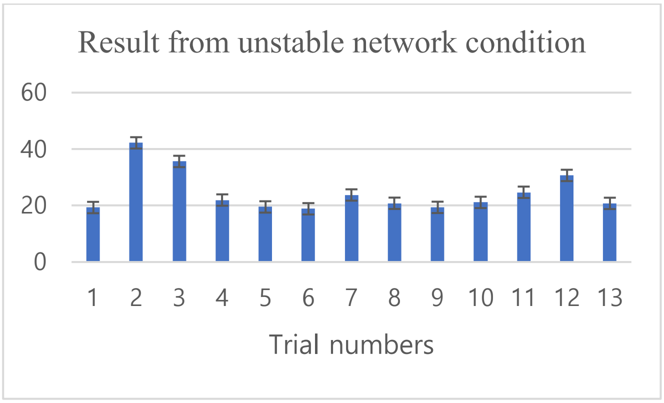

Figure 7.(512x512, turn : 1000, number of nodes : 4)

In conclusion, Although the tests conducted above have made various efforts to reduce errors in the process of obtaining results, there have been limitations that do not achieve stable results, as shown in Figure 7. The probable cause is not fast enough and the unstable network conditions. For example, when we opened the AWS instances in an environment where even web search has delayed and ran benchmark tests under the same conditions several times, very unstable results were derived, such as in Figure 6. Thus, we reckon that if we had experimented in a much faster and more stable Internet environment, the test would have produced more reliable results. Although it was not executed due to AWS credit constraints, it is estimated that a larger decline trend could have been observed with the number of walkers if the larger image and larger value turns were given, especially in the distributed-parallel part. In our futher research, if one of the servers leaves the connection among the local-broker-server, based on our implementation, the connection of the local-broker will continue, however, the value of the alivecells received from the server will not be keep received(all alive cells value will be 0) as the broker-server is disconnected.

In addition, if an error occurs while importing live cells to a broker from a server, the calculation that was temporarily assigned to the server that failed through another server is processed, and the server that failed is excluded from the slice that contains the server from the next turn. Based on this mechanism design, the failed server was excluded and the failed attempt was made to implement a fault tolerance that could be done separately on another server, but capturing an error of the server was not succeed.We tried to capture errors in all workers with functional literals continuously, but this way created fairly weighted overheads, and the errors were also not properly detected. Therefore, when the server was calculated, it received a signal through the channel and changed to check if there was an error here, but the error was not properly found until all the servers were closed. Meanwhile, there is one of distinct advantage of our code; other key presses working when the program status is pause. This is due to our distinct implementtion way that continuously captures the keypress with goroutine and for loop, even when the program is paused, hence the keypresses could be recognized when the state of the game is paused. In developing the implementation of various versions of the game of life, we faced a number of problems, and in the process of breaking through the problems, we gained deeper knowledge, and valuable experience. our last goal is to lead to success what we have not finally realized with the knowledge gained based on these failures

References

[1] Steven White, Kent Sharkey, David Coulter, Drew Batchelor and Michael Satran, Using Mutex Objects. Microsoft, 2021. [online] Available at: https://learn.microsoft.com/en-us/windows/win32/sync/using-mutex-objects [Accessed 29th November 2022]

[2] University of Bristol (Computer Systems A), 2022. [Online]. Available at: https://www.ole.bris.ac.uk/bbcswebdav/courses/COMS20008_2022_TB-1/CONTENT_2022/solutions/conc_lab1.zip [Accessed 29th November 2022]

[3] Intel, What Is Hyper-Threading?, Intel, 2022. [online] Available at:

https://www.intel.co.uk/content/www/uk/en/gaming/resources/hyper-threading.html [Accessed 29th November 2022]