Yumu Xie

po21744

Parallel & Distributed Implementation

Game of Life

Stage 1 – Parallel Implementation

1. Functionality and Design

1.1 Functionality Implemented

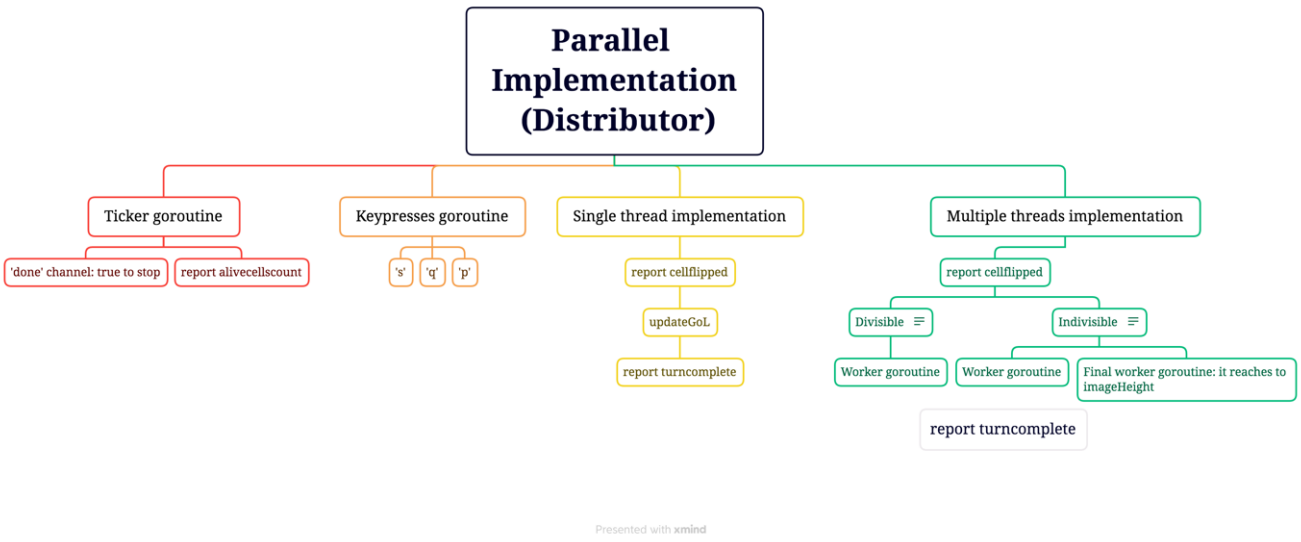
The first step of parallel implementation in GoL is to read initial PGM image. For reading PGM image, I used distributor channel to communicate with IO. I sent 'input' command and filename to IO by utilising command channel and filename channel, so IO can transfer byte back to distributor by using input channel and I save it to world by using two for nested loop to traverse the whole initial world which is a 2-dimensional slice with image height and image width. Then, I wrote a function called 'updateGol' in order to evolve world according to four rules of Game of Life. The 'updateGol' method uses module number way for considering the whole world. It ensures that it only considers the world and will not exceed to any other places. The 'output' world is created for storing the world which just gets updating once this method is called. Two for loops are nested for traversing the whole image height and image width. Furthermore, I used another for loop for traversing this list which contains adjacent eight points around target point. Then, applying GoL logic rules is what I did for rest of this method. After that, I wrote a for loop to traverse every turn. Each turn, a template world is created to save evolved result and world (the main world) is updated by replacing template world in each round. It covers the original data (original main world) and gets ready for next round's evolving. Also, I created a 'final' world to store the world state after all turns are completed. The 'final' world is used for reporting the final turn complete event. So, after all turns are finished, I directly let 'final' world is equal to world itself which is just outside the for loop of traversing all turns (all turns have completed). The second step is the hardest part I think in this coursework. I need to transfer it from single-threaded implementation to multi-threaded implementation. I tried to find idea from medianFilter lab work. I split single-threaded with multiple-threaded, so I used if statement to split it. For single-threaded, there is no worker only updateGol method inside because I just need to evolve (update) the whole graph. For multiple-threaded, I created a complete nil empty world (NewPixelData) at the start of each turn to append the result into this nil world. Besides, I also need to consider creating multiple channels for each worker. For each worker, it has its own channel to transfer part of graph which it is responsible for evolving that part back to distributor. The last thing I need to consider is the situation that image height cannot be fully divisible by the number of threads, which means the worker height. So, I used if statements to split them away. For those who can be fully divisible, they just do normal worker goroutine. For those who cannot be fully divisible, I let the last worker to do more things and I understand that this is a lazy way for doing this step. Finally, the nil empty world (NewPixelData) to append these data back. Each turn, the world gets update according to this world (NewPixelData) and gets ready for worker to evolve it in the next turn. Next turn, the nil empty world (NewPixelData) is created again and do things I illustrated above again. In the step three and step four, the ticker and the output PGM image, I used the ticker which is the built-in function in Golang (Golang has built-in time package for user to use, it contains ticker) and goroutine to start it concurrently. The 'done' channel is created as a signal for stopping the ticker. After the final turn is completed, I transfer true to 'done' and it stops immediately. For outputting image, I putted it after I report final turn complete. I give output command and send correct filename with turns. Then, I transfer 'final' world byte by byte to output channel to IO. In step five, keypresses operation is also a parallel goroutine to be started just after ticker. The keypresses channel can be called to use. During termination of program, I used os.Exit(0) (0 means success of termination of the program)

method to stop the program. For ‘p’ key press operation, I used mutex lock which Golang has built-in package for user to utilise to stuck after first ‘p’ keypress.

1.2 Problem Solved

In the first step, I got stuck in transferring correct filename to IO. The image folder hints me that I need to send 16x16, 64x64 and etc images to IO. At that time, I did not find out that the p (Params) has already specify what I need such as turns, threads, image width and image height inside GoL. Finally, I used ‘strconv’ method (‘strconv’ package used) to convert from int type to string type according to image height and image width of params and sent right filename to IO. For the second step, I mainly applied medianFilter idea into my coursework, so splitting is not the main issue. I faced the critical problem which is that how to design the new ‘updateGol’ method for evolving. For original ‘updateGol’ method, I used modules number way to let target point (cell) can visit adjacent eight points (cells). In worker, although the new evolving way I used is still modules number way, it has slightly different. For width, it is exactly same as previous, since I do not need to split width. For height, I need to plus startY in every judgement conditions, no matter the rules and the eight adjacent points. That’s the main bug I struggled when I was writing worker. I used to split wrong world. For example, my previous worker checked limited world (partial world) instead of the whole world. That makes the world cannot be updated correctly. In the step three, the main bug is that I put ticker goroutine after GoL evolving logic. So, it becomes that after evolving, the ticker goroutine starts ‘slowly’. During debugging time, according to TA’s guidance, I switched it to where I just received byte from IO by using input channel. It starts concurrently with the single-threaded and multi-threaded implementation. For step five, the main problem is in around ‘p’ key press operation. After the first ‘p’ keypress, the whole SDL and program need to be paused. At first, I tried to use channel to stuck it. Because of channel’s principle, if channel cannot get receiving, it will stuck at that point and do not proceed the program. However, I got stuck on how to use channel to stuck the program correctly for much time. It always reports bugs in some weird points. So finally, according to TA’s guidance, I switched to mutex lock. What I need to do is to lock it after the first ’p’ key press, and unlock it after second ‘p’ key press. Besides, I putted unlock and lock in the final output stage which I just received data from worker’s channel and append them to the complete nil empty world (NewPixelData). Besides, the world just gets update can be putted between locked and unlocked. It is still work I think. The mutex lock is ‘concurrent’ as well.

The following graph shows the overall logic of parallel implementation:

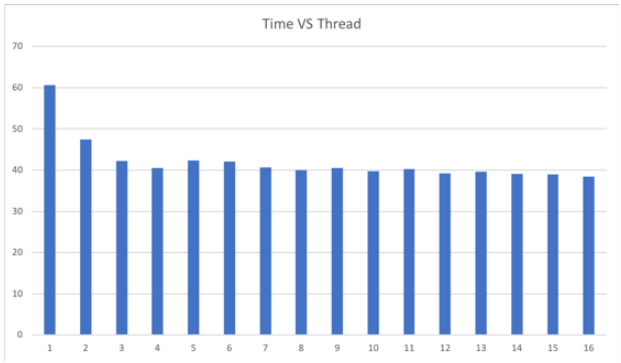


2. Testing and Critical Analysis

2.1 Acquiring Results

The benchmark tests were running five times on my own local machine. The CPU of my laptop is 4-core-Intel-i5-7267U with 3.10GHz. The overall time consumed to pass all benchmark tests is quite too long. It spends around eleven minutes (sometimes even more) averagely according to five times experiment. I took the mean number (average number) of these five tests results as result source and compared it with time (seconds) to plot graph (x-axis and y-axis).

2.2 The benchmark tests graph



In this picture, the y-axis is the time consumed to pass benchmark tests, which the unit is second(s). The x-axis is the number of threads my GoL program used during benchmark tests. Clearly, the general trend for this graph is decreasing from one thread to sixteen threads, which means that the multiple-threaded implementation is indeed faster than single-threaded implementation. However, there are still some outliers showed in this picture and I know that it is normal in real life. For example, when the number of threads is four, obviously it is faster than when the number of threads is five, but in theoretical, five should be faster than four because of more threads implemented in program. The basic principle here is that if I use more threads to implement my program, these threads do work concurrently (parallelly). So, the efficiency of working should be high, and time consumed to finish work should be short, or vice versa.

2.3 Thoughts of results

Even though the general trend of this graph is decreasing, which means that it seems like fine, the outlier cannot be missed for analysing. For thread four, the efficiency of working is almost close to thread eight. Actually, from the data I got, with the increase of number of threads, the efficiency of working cannot make a great progress anymore. It is getting to around forty seconds. The maximum threads (sixteen threads) reaches the peak performance which is around thirty-eight point five seconds. In my opinion, the increase of working threads can actually improve performance of working, but sometimes in some special circumstances, for instance thread number is four, the performance is even better than situation which thread number is five or six! That means that in specific number of threads, the performance is good enough. In other words, the more working threads used is not equivalent to better working performance. In real life, some conditions such as room temperature, operation system of machine, code optimisation, logic consistency and even compatibility between hardware and software can indeed affect or constraint the performance of program. Without very careful contrast experiment, it is hard to test it specifically.

3.Conclusion

Overall, in order to make a full use of CPU, more working threads used is necessary. However, enabling more working threads means more consumption of enabling threads, including time and performance consumption. CPU will also take much stress or pressure because of more

threads. The hardest part of parallel implementation is splitting single-threaded to multi-threaded I think. There are many things I need to take a serious consideration, such as whole graph consideration and indivisible worker height. That's what I learnt from parallel implementation.

Stage 2 – Distributed Implementation

1. Functionality and Design

1.1 Functionality implemented

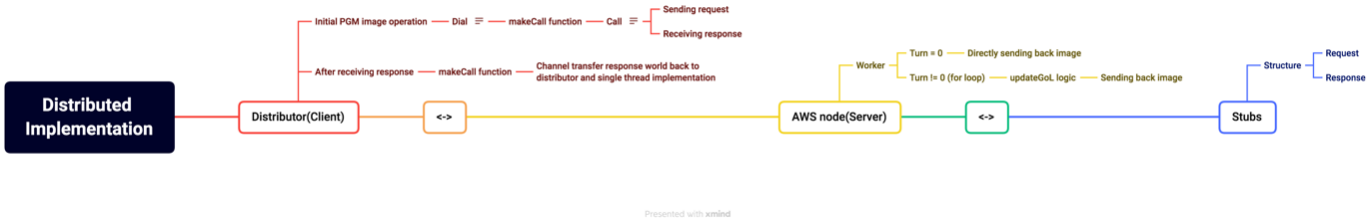
The first step, in other words the basic step, is to implement single-threaded implementation between client (distributor) and server (AWS node). I generally took the idea of secret string lab work and used some code from solution of secret string directly. The key idea of doing step one in distributed implementation is to achieve that there is no GoL logic in client. I transferred world which needs to be updated from client to server and got updated world back. Server receives world and params, then it proceeds the turn. The second step is difficult as well when I was doing. How to simultaneously do ticker in client and send signal to server to do corresponding operations becomes the main problem here. I start the ticker goroutine just after client has been made and dial server. The basic structure of this goroutine is similar to what I did in parallel part. For every two seconds, I call NewWorker after I create NewRequest and NewResponse. Then I directly report response turn and response cell to AliveCellsCount. Back to server, I made a NewWorker to deal with this request. If the request is true, I send true to a global variable channel which called signal. For Worker, I start a goroutine before normal processing turns compared with step one. If it receives signal and it is true, Worker will send current completed turn and number of alive cells of current world back to global variable channels which called turnChannel and cellChannel. These two channels will help NewWorker to get the data it wants, and NewWorker will send back these data immediately. Unfortunately, after I start the server, if I consecutively run single-threaded test and alive test, the alive test cannot pass. If I run them separately, they both can pass. The mechanism of why this problem causes I think is that server keeps the last state it gets, when I was running single-thread test, ticker goroutine starts automatically and server is indeed processing it. When alive test starts to run, the state that the server was processing did not get clean, so that the state keeps inside the server and report wrong completed turn and number of alive cells. For step three, it is almost same with parallel part. I put it after makeCall method and FinalTurnComplete. After all turns have processed, the final state should be outputted as a PGM image. I didn't finish step four because I almost run out of time. The 'q' operation of key press really struggled me, which asks me to clear the state of server without closing it. The logic of step four I think is that I create a goroutine for key presses in distributor (client) as well just like parallel implementation. Besides, it is in an infinite for loop and I receive 'key' from distributor channel of key presses. For each specific key press, I do corresponding operations in server.

1.2 Problem solved

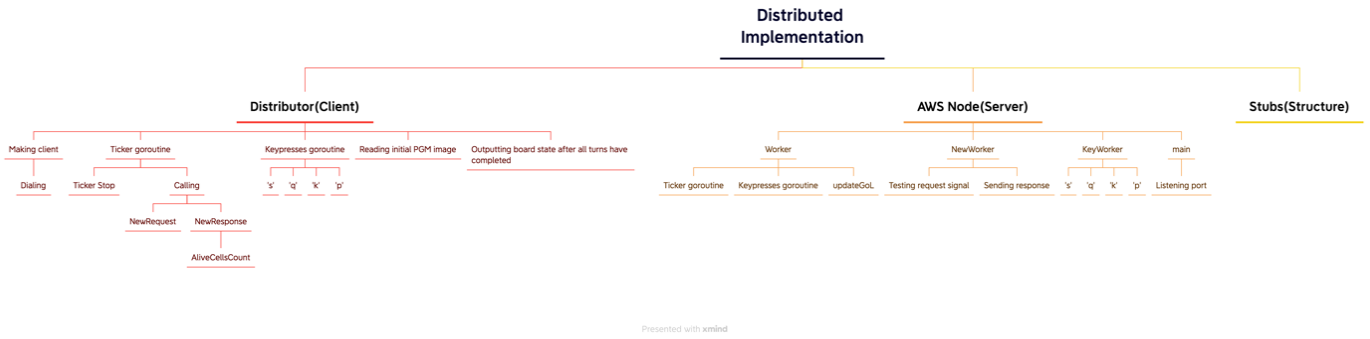
In first step, I got confused about how to make client, especially when I was using flag to create it. Using flag, I cannot pass single-threaded test, since flag cannot be redefined. The 'flag' method I used is also from secret string's idea. Eventually, I utilised directly 'rpc.Dial' method and defined server as a string with IP address and port number. For step two, I got stuck for long time. At first, I thought that I need to create two goroutines at both server and client for sending request and receiving response. But, actually, I cannot ensure that these two goroutines can start simultaneously. So, that is wrong logic before. After that, I tried to think that maybe I can use something like signal to notify server to send back completed turn and number of cells to client during normal processing turns. The communication of two workers struggled me for some time. I guess I need to use some global variable to make a connection between these two workers. When I was making global channel, I used wrong syntax ':='. Actually, I need to use 'var' to make global variable. That's also a problem that wastes my time. In client, I used to utilize channel to

receive response, though finally I directly call them in ticker goroutine. I used unbuffered channel so that it always gets stuck at some point. After I changed it to buffered channel, it finally works and is good for running. For step four, the main bug I faced is in stubs. In stubs, we need to create global variable for each worker and name it like “GoLWorkerOperations”. However, I named a wrong name (“GoLKeyWorkerOperations”) in KeyWorker. So, client cannot make a connection to server. TA helps me with it by using ‘err’ and print it out if err is not nil. The ‘rpc’ will report that it cannot find server.

The following picture shows the step one logic of distributed implementation:



The next graph shows the step two & step three & step four logic of distributed implementation:

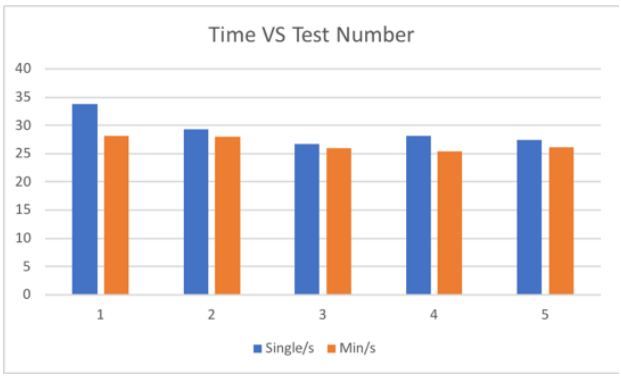


2. Testing and Critical Analysis

2.1 Acquiring results

I still manually run benchmark tests in intelliJ idea for five times on my own machine. The CPU of my laptop is 4-core-Intel-i5-7267U with 3.10GHz. For distributed part, what we need to focus on is the connection between server and client instead of multiple-threaded implementation. So, all results I obtained are single-threaded. I also tried to use mean number to calculate a reasonable result out. The average time spend to pass all tests is around seven minutes, which is quite surprise for me. The constrained conditions is that I ran them in local host, which is on my local machine.

2.2 The benchmark tests graph



The y-axis means the time spend on passing benchmark tests, and the x-axis means how many times I did the benchmark tests. The blue bar is the first single-threaded test result. The orange bar is the shortest time to pass benchmark test and it is obtained from other threaded tests results. Averagely, the blue bar, which is the single-threaded test, spends around twenty-nine point one seconds. The mean time of orange bar, which is the minimum time, consumed about twenty-six point seven seconds. Obviously, in blue bar, except for the first-time test, the second, the fourth and the fifth time tests are almost the same, which is around twenty-eight seconds. The minimum time to pass tests is the third one, which is almost twenty-six seconds. The orange bar (minimum time spend on benchmark tests) shows more stable trend compared with the blue bar (single-threaded bench mark test).

2.3 Thoughts of results

These test results are subject to Internet and CPU. If the speed of internet is slow, the results will take much time to send and receive for both client and server. For server, CPU is more significant because I put GoL logic inside server instead of client. Server becomes the main component to deal with evolving of graph. For client, it needs to ensure that sending and receiving information are good.

3. Conclusion

The most impressive aspect of distributed implementation is that I should be very careful about the structure of stubs. The structure of stubs is familiar for me. In my view, it is similar to interface in Java. But, they are not identical things. Stubs have strict requirements of naming and making structure. Generally, distributed implementation helps me a lot about understanding the connection between server and client.