The Game of Life

Hongjie Wu: cu19430
Fanzhaoyang Lei: vr19825

1 INTRODUCTION

This project is based on Conway's Game of life, realised by golang and given skeleton, we simulate the game in two ways, it could either be run on a single local machine or using the game controller on the local machine and calculate with serval nodes on the remote machine.

The rule of Game of Life is easy, given a binary picture, with the initial state and during the running turns, we need to calculate if each cell is alive or dead in the next turn.

- The cell dies if it is a live cell with fewer than two live neighbours or more than three live neighbours.
- The cell becomes alive if it is a dead cell and there are exactly three live neighbours.
- The cell is unaffected if it is a live cell and with two or three live neighbours.

OVERVIEW OF THE PROJECT

We use two ways to simulate this game, parallel and distribute. In the parallel stage, we mainly focus on how to implement the logic so that the transformation in each round could be visualised through SDL. Later in distribute stage, we use AWS as a server for calculation, in this way, the local machine could receive the result of alive cells and rounds shown on local SDL window. To analysis, we write benchmark tests for certain turns and thread.

2 TWO MAIN STAGE

2.1 Parallel Stage

Methodology

- **Parallel Calculation:** We divide the project into smaller-scale ones, to be specific, in our project, the world is split into parts and given to serval workers with regards to the number of threads. In the sequence, the parts would be merged which would be used in the calculation for the next turn.
- **Discover Common solutions:** During the working flow at this stage, we try to avoid multi-use of the same code and generate the acceptable code for all the circumstances. For example, the application of workers and the extracted functions in *gamelogic.go* file in *gol* package.
- **Data Visualisation:** After our work in coding, we make two CSV files with different output from the benchmark, with *seaborn*, we analysis the output with the line plot generated.

Corporation between Goroutines

At this stage, there are two places with essential goroutines.

- At the beginning of the project, we use goroutines combined with channels to establish communication between *distributor* and *startIo*. To retaining this communication safety. We use common channels and leave the channels inside the *distributor* and *startIo* structure one-direction only. Inside each goroutine, there is a select case, which equipped the goroutine to give different actions. For example, *io* would choose *readPgmImage* or *writePgmImage* in respect to what *ioCommand* has sent. Meanwhile, the visualisation on the SDL window is also achieved by goroutines, during the running of *distributor*, events the *distributorChannels* received are sent to SDL.

- Inside the *distributor* function, the goroutine is implemented on a worker which is used to do the partial calculation. The range of height and width is given to each worker, as after calling *calculateNestStage* function inside worker. Afterwards, the new parts of the world are appended into an array of channels. Later, the weaved new world with the implementation of a for loop, *mergedWorld*, is considered as the new calculated world.

Implementation of multiple workers

To keep the running time of the programme as small as possible, as what has been discussed before, we use multiple workers to calculate at the same time. As the number of threads is given at the first stage, to use them at maximum level, we use the number of threads to divide *p.ImageHeight* which is the *HeightPerThread*. As the division calculation in golang is default as rounding floor, we don't need to consider the overflow problem here. We use a for loop to iterate the number of thread minus one time, considering that height is not divisible by the number of threads, one worker is left outside the for loop to handle the rest of picture all the time. This method is suitable all the time.

Working Flow at parallel stage

After entering *main.go*, the SDL window would start at the same time with calling the function *gol.Run*, where the common channels are created, so as the two goroutines, *distributor* and *startIo*. The *ioCommand* is used to notify the state of the current programme and the rest are used to pass variables between two goroutines.

Generally speaking, when *ioInput* is transferred to *distributor*, before all the calculation, the initial alive cells would be flipped. In the sequence, there is a for loop to contain all the calculations with

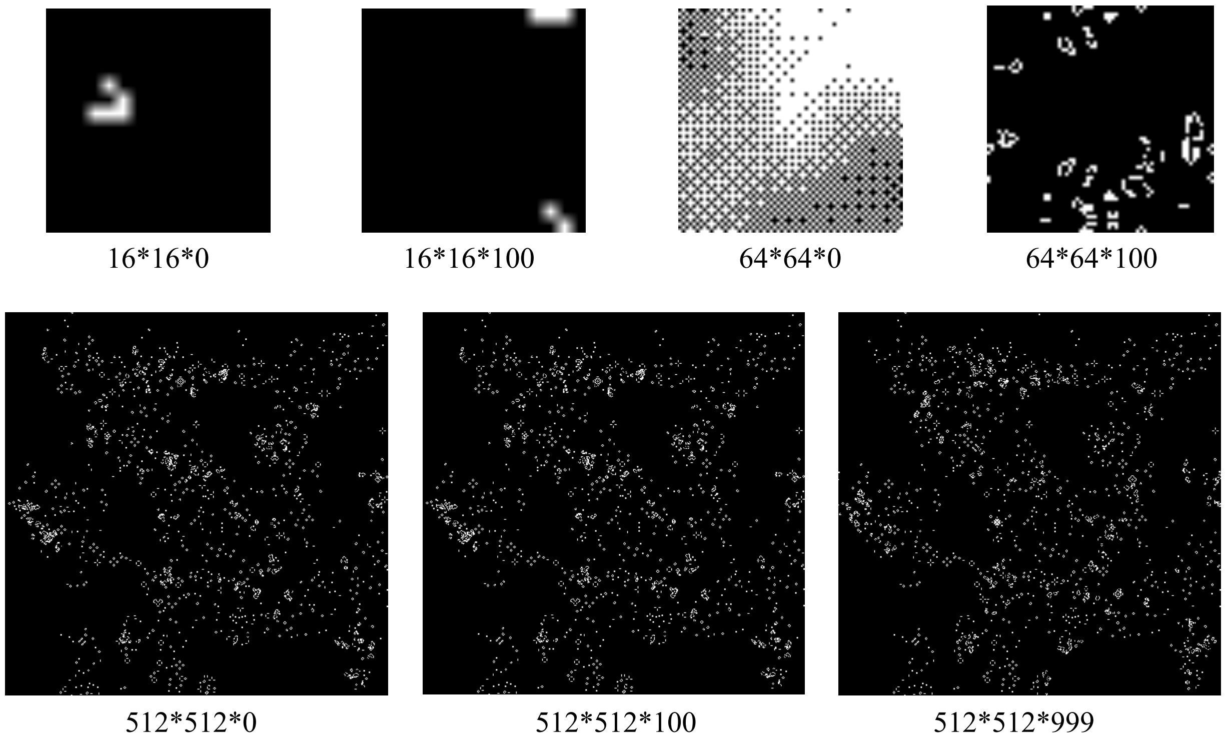

Fig. 1 Images after Certain Rounds of Calculation

multiple workers to calculate, starting by making the calculated world immutable. Meanwhile, there is a *select* after which is corresponding to distinct receiving variables from the *distributorChannel*. The first case is using a ticker to send the alive cells every two seconds and the second case corresponds to the SDL controller. After all those steps, *outputWorldImage* function would be called, subsequently, *ioOutput* would be sent to *ioCommand*, the *writePgmImage* would output the shot of the current world with the title which contains the height, width and the number of thread.

Figure 1 illustrates some chosen PGM file, where we could see the transformation after calculation. It seems that we could easily tell the difference of the pictures with smaller sizes, with the comparison between the 16*16 and 512*512.

Visualising with SDL Window

The SDL controller is added at this stage, there are three keys related to the controller.

- **s:** The *event*, *Executing*, could be passed to the *event* channel in the *distributorChannels*, generating a shot of the current world.
- **p:** The calculation would be paused until the next **p** been pressed, an event *Continuing* would be sent to the *event* channel and output a picture of the current stage. During the waiting time, neither **s** nor **q** could re-activate the process.
- **q:** The *event*, *Quitting,* is passed to the *event* channel, the process of calculation terminates and the final picture is generated.

Analysis with Benchmarking

For the analysation of the performance of the first stage, we use a *Benchmark* function inside the new file we created named *benchmark_test*. To test the maximum performance, we only use the 512*512 graph and set the turns from 100 to 5000, however, the number of threads is from one to sixteen gradually.

In figure 2, we compare the running time in different numbers of threads. The number of threads could impact on the running time. What surprised us is that the trend of the running time is not impacted by the increasing number of turns. The running time peaked when there are two threads and 400 turns ran, meanwhile, the fastest running time is when there is only 100 turns with eight workers. Fewer workers won't always lead the programme slower, and the increasing turns do not grow the running time.

Take the lowest point as an example, 512 is divisible by eight, in this way, each worker would take the equal pieces of the world to calculate. Since the goroutines are working together, the running time for every worker is roughly the same, the merging step doesn't have to wait for each single worker. Although two is also divisible for 512, each worker still needs to calculate a large world, which sometimes might not always be beneficial.

There are also some other abnormal points. For instance, when five threads running for 100 turns and seven-thread running for 5000 turns, the running time massively increases. This might because of the happening of waiting condition, where those finish earlier has to wait for the longer ones.
Figure 3 shows the running time of this programme with the same round on pictures with different sizes and numbers of threads. Typically, the running time decreases when the number of thread increase, which is sensible that, now a worker only needs to calculate a smaller part. The running time reaches the maximum when eight workers are running on the 512*512 picture. Considering the

other conditions of the same picture, this point might be the abnormal point, when the computer might reach a point where the calculation ability reduces.

The trend of the 16*16 running time is not like what we are expecting. The running time for two, four and six workers is far more than the rest. Take the six-thread running time as an example, except the last worker, each worker only receive the one with 2*2 pieces, as a result, the last worker would do the rest 4*4 world. This is the same size as what the four-thread workers receive. We could suggest that the running time for 16*16 graph is partly defined by the size of pieces each worker received.

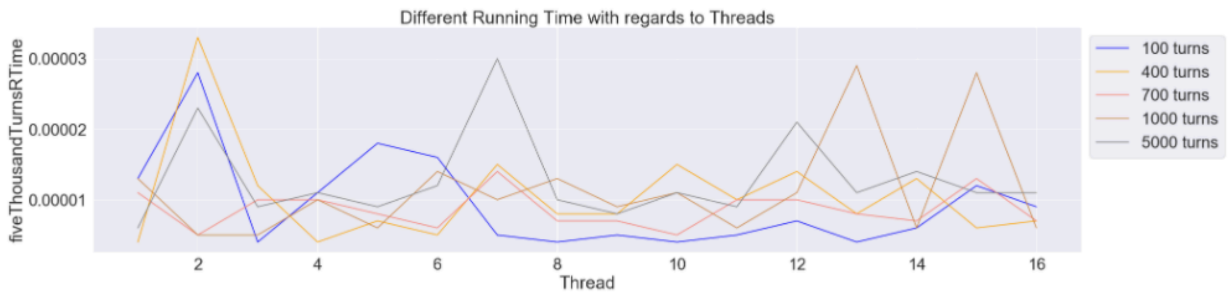

Fig. 2 Different Running Time with regards to Threads

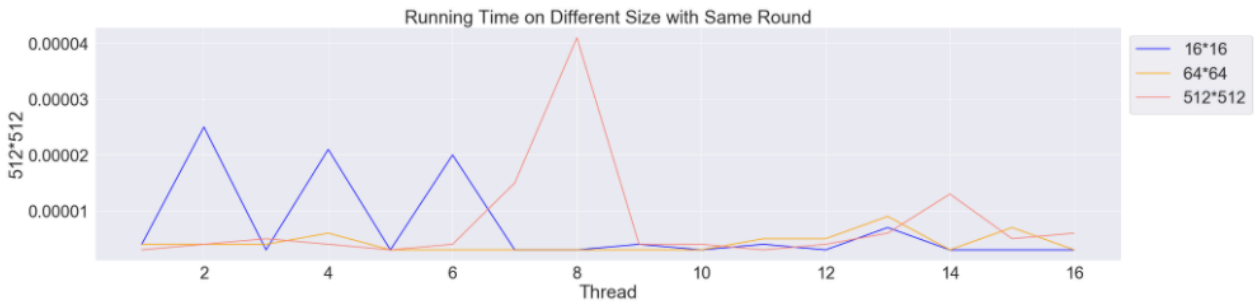

Fig. 3 Running Time on Different Size with Same Round

2.2 Distribute Stage

Methodology

- **Divide and Conquer:** Like what has been done in parallel stage, we are trying to break the larger problems into smaller ones. This is at the stage when trying to generate calculation in multiple nodes. Moreover, when designing the distributed system, at the first step, we extract the controller from the original game logic and then trying to run it on a local machine before using AWS to achieve.
- **Discover Straightforward Solution:** At some stage of this part, there might be other solution using for loop to achieve, which seems to be clear at the first glance. We still chose the unfolding approach, as the matter that, most of the conditions, the number of these loops are defined and small, meanwhile, it would be easier to understand the logic behind.
- **Distribute Calculation:** After dividing the problems into pieces, the calculation is accomplished in the different IP address. Although we are using one computer to run multiple visual machines at this moment, in the future, we could connect to other real computers, ignoring the physical wires.

Working Flow at Distribute Stage

From figure 4, it is obvious that, besides the *io*, *distributor* and *SDL*, to manage the computation and adding flexibility, two other files are added to simulate the server and sub-server. And in our programme, they are *broker* and *factory*. In this way, after *distributor* receiving the initial world, we would call the RPC function, *GetQStatus* to detect if q is pressed to quit the game controller and whether to read a new PGM file or continue the legacy game.

Soon after, the distributor would send each factory the work they need to finished with the RPC function *CalculateNextTurn*. Implementing the related functions such as *CalculateNextStage*, the calculated parts would be sent back to the *broker* and merged world in broker would be posted to distributor for further turns calculations or SDL visualisation.

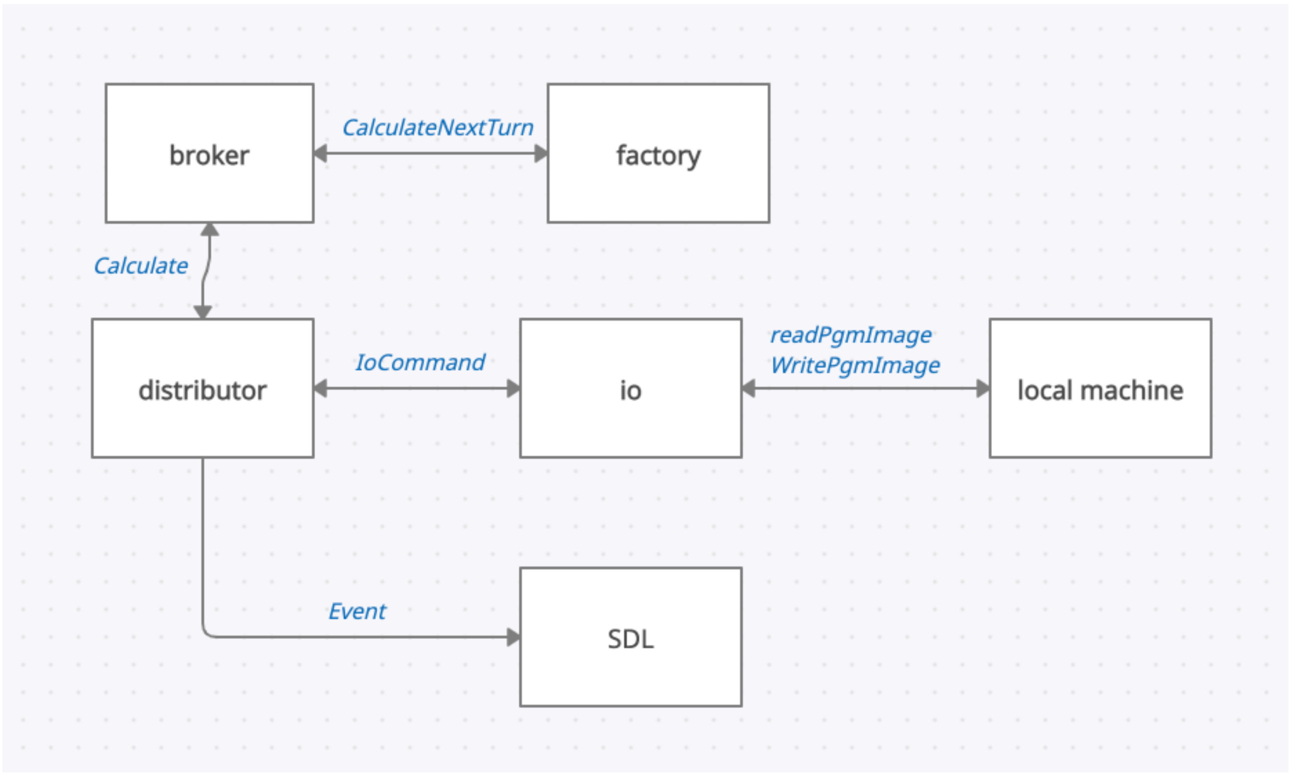

Fig. 4 Main Working Flow in Distribute Stage

SDL Controller

Compare to the SDL controlled in parallel stage, there are two differences.

- **q:** When **q** is pressed, the controller closed as what it is in parallel stage, however, the data in *broker* should be conserved, later when the controller starts again, the *broker* should continue calculating with the legacy data.
- **k:** Pressing **k** means that all the *factories* and the *broker* should be exited. When **k** is pressed, it would call the RPC function, *QuittingBroker* which is in the *broker*. Inside *broker*, this function could call the RPC function in the *factory*, *QuitingFactory*. The later one would use *os.Exit(0)* to quit all the servers.

Transferred Data

To kept the speed of transferring data fast, we set some global variables inside the *broker.* What is important for the *distributor* is the changed cells and calculated world, and the *broker*, which is considered as a server did all the calculation, hence, in each turn, the *distributor* only send the uncompleted turns to the *broker* to reserve.

Meanwhile, between broker and factory, only the ranges of height and width and the copy of the world on the broker are necessary. It is plain that the ranges guarantee the right result, and the copy gives us the capability to decide should the cell be flipped in the local machine, with regards to the function *CalculateNeighbors*.

Missed Data

When running the distributed programme, we find that the change on the SDL window is really slow, comparing to when running the parallel programme. Digging to our structure, this limitation might due to the transferred data in each round and the ways of communication established.

Although we use the global variables in *broker* to prevent sending plenty of data. Between *broker* and *factory*, dispatching huge amount of data is not prevented. While a weak network connection is one of the factors, we assume that no data is missed during this transfer. Additionally, the calculation and visualisation occur concurrently, some data collision might appear.

3 CONCLUSION

Achievement

In a nutshell, we use two different approaches to reproduce the Game of Life. Now, we could run on a single machine with a parallel stage or run on serval machines with a distributed stage. Simultaneously, we use goroutines and channels to communicate between different functions, giving easier access to exchange the variables.

Limitation

Owing to limited knowledge, some details haven't be considered earlier and become a legacy problem. For example, before using the structure like the *broker* and the *factory*, our assumption is extracting the calculation to a server. Later we change the server into *factory* and add a *broker* between the *broker* layer and *client* layer. As the calculation function used in the *factory* is the same as that been used at the parallel stage. The unused *Param* has to be imported as a parameter, leaving the code not neat.

Future Improvement

Before improving the performance, parameters of functions need modification. One of the significant examples is *Params* in *CalculateNextTurn* function. With the solid foundation of the coding structure, we could use both parallel and distribute structure. After giving each node the area to calculate, inside each node, there could be various workers to perform. Thus, if we benchmark before deciding the area each worker should take, the best performance would be accomplished.