REPORT FOR CSA Coursework: Game of life

Yifei Wang & Zexiang Jin

gf20334 & kg19065

Stage 1 – Parallel Implementation

-Image Reading and Saving

Image file reading occurs during the program initialization phase, and image saving occurs before the program ends or when the user presses the s key.

io.go provides the ability to read and save files, which are controlled by sending commands and data through channels. So when the program starts, I create 3 channels :***ioFilename, ioOutput, and ioInput***, where ***ioFilename*** is used to send the command to be read. ***ioOutput*** is used to send file data to be saved, and ***ioInput*** is used to transmit file data to be read.

The process of reading the image file:

1. Send ***ioInput*** through the ioCommand channel.
2. Use the ***ioFilename*** channel to send the file name to be read.
3. Read file data into 2D slices through the ***ioInput*** channel.

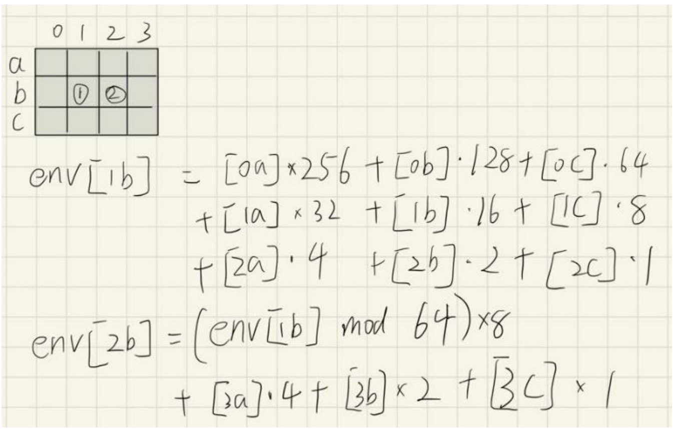
The process for saving the image file is similar:

1. Send outInput through the ***ioCommand*** channel.
2. Send the file name to be saved through the ***ioFilename*** channel.
3. Send 2D slice data through the ***ioOutput*** channel.

-Logic for the Game of life;

At the beginning, we did not pay attention on optimizing efficiency, but pursued stability, so we copied the code for Game of Life from first weeks work, we first traversed every point on the picture, and then traversed the survival state of nine cells around each point to judge the next survival situation of this cell. This logic worked very well, but when we finished the work and started optimizing. We found that when the image was large enough, this traversal affected the performance of the project. When the graph is 512*512, the function first needs to traverse 512*512 points, which is unavoidable, but we still need to traverse each point before we can finally confirm the survival state of this point, which is a waste of time, so we wondered if we could optimize it, and we came up with a method. We use the method of image interpretation (Fig 1.1) to calculate the environment value of each point, so that we can directly update the environment value of other points except the first point， according to the formula in the figure, reducing the number of reading two-dimensional array 6 times. The number of points traversed by the previous method is 2,359,296, and the number of points traversed by the improved method is 789,504, so we can see that we save two thirds of the time. Our conclusion is that there is no significant difference in the small 16 by 16 image, but as the image traversal increases, the time saved becomes larger and larger.

Then there is the limitation of our method. Our method is only useful when we calculate a large number of data, because we are allocating work from one thread to one line, which limits our allocation of work. For example, when the number of threads is greater than the number of lines, our method will reduce the efficiency of work.



$$env[1b] = [0a]\times256 + [0b]\cdot128 + [0c]\cdot64 + [1a]\times32 + [1b]\cdot16 + [1c]\cdot8 + [2a]\cdot4 + [2b]\cdot2 + [2c]\cdot1$$

$$env[2b] = (env[1b] \bmod 64)\times8 + [3a]\cdot4 + [3b]\times2 + [3c]\times1$$

(fig 1.1)

-Goroutines used and how they work;

At the beginning to parallelize the Game of Life, we intended to use channels to transmit data to avoid deadlock. We created several channels to transmit whether the threads are done (***workdone*** *chan bool*), how each line will looklike in the new turn (***newTurnLine*** *(chan []int)*), and whether we need a new thread (***needNewWorker*** *(chan int)*), we then create a goroutine function called ***Worker***, which contains the specific tasks we want it to do, such as:

1. It needs to complete the task specified by the passed parameter (row int).

2. Because we don't want the main thread (receiving task) logic to be too cumbersome, we expect the worker to decide whether to create a new thread by itself.

3. Allow for the fact that when the image becomes large and all threads are working, use a recursion idea to continue the task.

Next we create a function called ***waitWorkerDone***, which blocks the code from running and receives messages from ***Worker***, it implementing the following functions:

1. It needs to update the number of threads and terminate the function when no thread is working and return the result.

2. We want to leave the job of creating threads to the main thread for debugging purposes, so we need to create a new thread after receiving a message from the ***Worker*** function.

3. After each worker is calculated, each worker will send its processed results, and the main thread needs to integrate these results.

Next is what we think we did well and what we didn't do well when designing goroutine:

-Advantage

·Convenient debugging

-Deficiencies

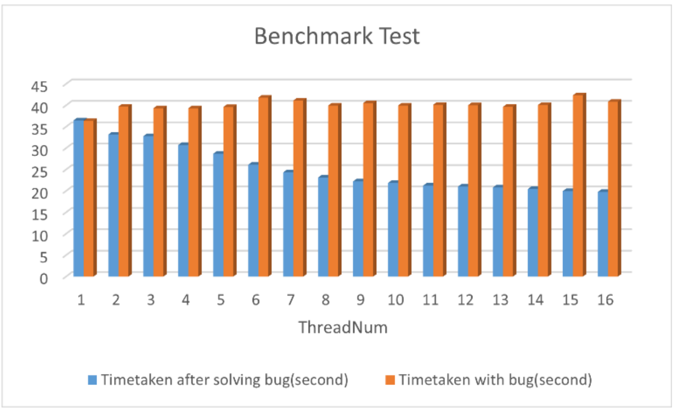
With recursion, the efficiency is lower than with traditional traversal, and with recursion, the loss of efficiency is more obvious when the number of threads is large enough and the workload is large enough. If recursion exceeds the maximum length of the stack, memory will overflow and the program will report an error

We managed to optimize the recursive part of our code by using thread pool thinking before committing. In addition, the latest code creates all the threads at once and continuously fetches tasks from the thread pool, which makes our code run much more efficiently. The specific optimization is shown in the benchmark partition.

-Benchmark

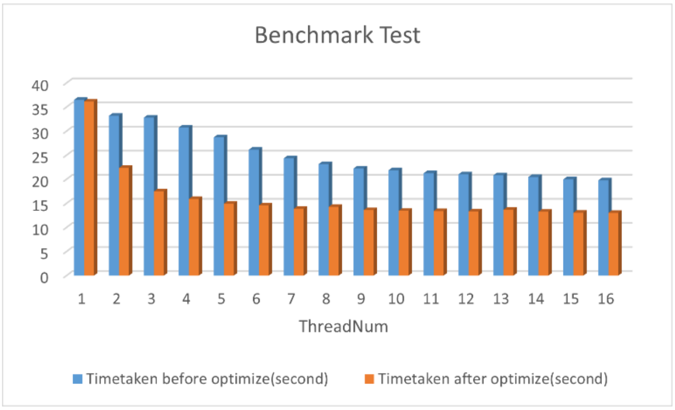
At the beginning of testing benchmark, we found that a single thread runs the fastest, which is very abnormal. At first, we thought it was because a single assignment consumes very short time, which makes the cost of creating and recycling a thread is larger than the resource of a thread processing its assignment. However, we soon realized that this was due to a logical bug in our code.
Once we make the logic right, multi-threaded tasks were as resource-efficient as we expected, 16 threads are 16 seconds faster than a single thread.

(shown in fig1.2)



(fig 1.2)

Although we had done the parallel implementation, we later found a better algorithm to optimize our code. We adopted a thread pool approach. Though the time for a single thread was about the same as that for recursion, staying around 36 seconds, but then as the number of enabled threads increased, The time taken by the application is significantly reduced compared to the previous time. We observed a significant performance decline as the number of threads created increased from 1 to 4,.After that we keep increasing the number of threads.(shown in Fig 1.3) ,although the overall speed is still faster, the optimization trend is negligible, considering the resources spent on thread creation. We decided that creating four threads was optimal unless we expanded the image area (for example, using 5120 by 5120 images)



(fig1.3)

Stage 2 – Distributed Implementation

First, we decided to create a connection between the server and the client via the RPC, and placing the Game of Life on the server for local remote calls. This will incur network communication overhead, but will greatly reduce local performance consumption. We plan to transfer the image area that the server is responsible for to the server. To maximize performance on the server side, we also ported the multi-threaded part when we ported the gol method, so the client also has to transmit to the server side, which requires how many threads. This will allow our program to perform at maximum performance.

The above is the main logic of our distribution: the main function creates multiple threads to call the ***initialSend()*** function and starts sending the part they are responsible for to the server address , then the server starts processing the image, at this moment the main function code blocks and waits for all the servers to finish processing to return the value and then outputs the subsequent content.

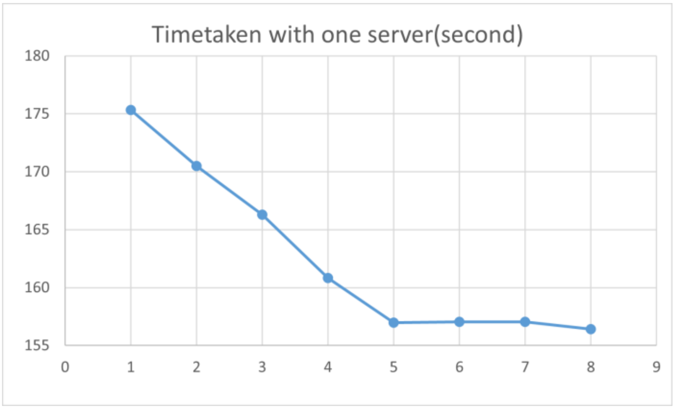
What we can do to make up for it: Although we implemented a multi-server distribution, each server needed to configure the port information individually, which caused considerable rework in the case of a large number of distributions. We tried our best to improve in this area, but we failed.

-Benchmark

Compared with parallel, we completed the code for distribute very quickly and successfully passed all the tests, but the performance loss is extremely serious. By debugging the code we discovered the parts of the distribution that seriously affected performance. In the parallel part, we found that the output picture and the input picture consumed a lot of time, and the larger the image, the greater the impact of these two parts. This is because the input-output image has to go through every point on the image. This is unavoidable in parallel, and it is the same in distribute. What we can avoid is not flooding the entire image. This made the program run so inefficient that even after 5 minutes in the 5120x5120 extreme test, the program still couldn't render a full turn. This forced us to shut down the application and start optimizing it. We worked out the following optimization directions: first by reducing the size of the images transmitted on the network; second, we tried to transfer only the list of living cells to the server to self-restore into a two-dimensional array; finally, we tried to cache the map that has entered the next round on the server. Unfortunately, due to time constraints, we only got to the first step. We pre-allocate the number of rows that each server is responsible for computing (starting rows, ending rows) and calculate the number of rows needed to loop the image (upper limit edge, lower limit edge). These parts are assembled together to save the size of the two-dimensional array that each server needs to upload. (Upper limit edge - main body - lower limit edge)

In order to improve the efficiency of run score comparison,although we can complete 5120x5120 512x512 image calculation by running the server locally, it takes too much time, so our run score will be concentrated on 16x16-500 and 512x512-100. And since we only created two aws, we can only see how two servers improve over one server, even though we have proven that multiple servers can be distributed.

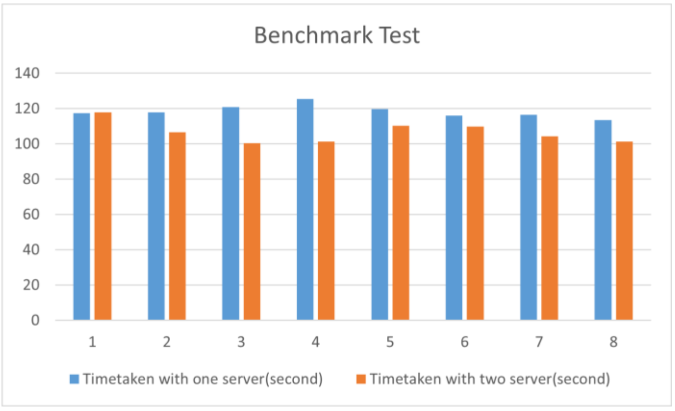
First we wanted to know that for the server side, opening how many threads is the most effective choice, so we ran 512x512-100 on a server, and the running score image is as follows(fig2.1),



(fig2.1)

We found that when the number of threads reached 4, the decline began to decrease significantly , so we conclude that the server is most efficient when it has four threads open.

We found that although the output time on the server almost doubled when 16x16-500 was running on two servers, we could not find a good effect on the client benchmark. We inferred that our main program was coupling too much with the server, which resulted in the increase of network transmission resource consumption. And it takes time for the image to transfer to the server. The final result is as follows(fig2.2)



(fig2.2)

Stage 3 – Conclusion

We finally reflected on our entire process and found the following advantages and disadvantages:

-Adantage

We finished the writing of the basic functions very quickly. Although the follow-up review showed that there were many deficiencies in the program, if we spent too much time on the completion of the basic functions, it would be impossible for us to optimize gol, parallel and other areas, and we clearly knew that there were still many directions to optimize. It just couldn't be done for time reasons

-Deficiencies

Although we saved a lot of time in code writing, we rarely used code management, and usually copied and pasted code synchronization directly, which wasted a lot of time in development. Besides, because we were not good at using git, we encountered a lot of frustration when uploading the server-side code to aws at last, which wasted time. If we save that time, we can optimize even more of our code.