The Implementation of Game of Life

Jee Lee dm24602
Haoxi Zheng rl23914

Parallel Implementation

1.Functionality and Design

In the initial stage of designing and implementing Game of Life, we focused on the overall architecture and structural framework of the project. The basic concept is to define the existence state of cells in space through grid modeling and store the current survival state of each cell - alive or dead. Then, we created the survival rules of the game through defining the logical conditions that determine whether the cells can continue to the next turn, thereby achieving the simulation of cell evolution across continuous iterations. The progress of the game is iteration-driven. Each cycle generates an updated grid structure, which leads to continuous changes in the visual presentation.

To ensure that all cell states are updated simultaneously during each iteration which is a requirement for accurate simulation, we addressed the challenge of large-scale, fine-grained parallel implementation by using different goroutines and channels, based on what we learnt before. This approach not only enhances computational efficiency but also ensures thread-safe operations. By distributing multiple goroutines across available CPU cores, we achieved parallel processing, which significantly improves performance and code conciseness compared to single-threaded serial execution.

1.1 Main goroutine - distributor

In our code, the main goroutine serves multiple purposes, acting as the distributor, which plays a central role in orchestrating the activities of all other goroutines. It initializes the entire project's runtime environment. Besides initializing the world, it also initializes the channels. In the distributor function, we create the “results” channel, which can receive computation results from the Worker. And we also create the “events” channel, through which “StateChange” events can be sent to inform the outside that the current state is running. Additionally, in the distributor, we start the timer goroutine.

Moreover, it can also implement the turn iteration of the GOL. Through “the for turn < p.Turns” loop, it gradually pushes the evolution of the Game of Life. Through the “select” statement, it listens to events. Corresponding to step 5, the select statement can listen to the KeyPresses channel and respond to instructions for three types of keys. When the "s" key is received, the distributor copies the current world state using the mutex “mu”, saves it as a PGM file using “savePGM”, and then sends the “ImageOutputComplete” event through the events channel. Upon receiving the "p" key, it changes the paused state and sends the “StateChange” event through the events channel. During the pause, no new Workers are started, but the ticker and keypresses listener can still be used normally. When the "q" key is received, the main goroutine stops the timer using “ticker.Stop()”, closes the timer through the done channel, and then waits for the Workers to complete the current iteration to finish the current round of calculations. It sends the “FinalTurnComplete” event through the events channel, saves the final form using “savePGM”, sends the “StateChange” event, changes the state to “Quitting”, closes the events channel, and the program terminates.

1.2 IO Goroutine

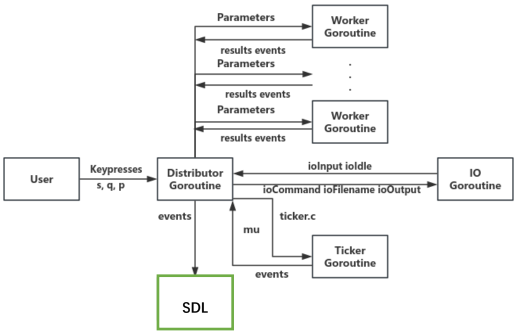
It is responsible for managing the reading and writing operations of PGM images. In collaboration with the distributor, it receives commands like "ioInput", "ioOutput", and "ioCheckIdle" via the ioCommand channel. Communication of file names and world data with the distributor occurs through the “ioFilename”, “ioOutput”, and “ioInput” channels. After task completion, it sends the I/O status back to the main goroutine using the “ioIdle” channel.

1.2 Worker Goroutine

It is responsible for computing the next turn's state of the world grid in parallel. The main Goroutine will start “p.Threads” number of Workers to carry out this task, and these Workers will collaborate with the Distributor Goroutine. The main Goroutine will divide the entire world into several sub-regions by rows, with each region having a height of sliceHeight = p.ImageHeight / p.Threads, and each will be handled by an independent Worker. After each Worker completes the calculation of the next turn's state within its corresponding sub-region, it will send the result back to the main Goroutine through the “results[w]” channel. Subsequently, the main Goroutine collects all the local results returned by the Workers and integrates them into “nextWorld”, which serves as the global world state for the next moment. During the calculation process, if the cell state flips, the Worker will also send a “CellsFlipped” event through the events channel to notify the specific situation of the state change in real time.

1.3 Ticker Goroutine

The Ticker Goroutine assists the background coroutine. The statistics logic is triggered by “time.Ticker”, and the shared variables world and turn are protected by the mutex “mu”. After the statistics are completed, the “AliveCellsCount” event is sent through the events channel to notify the number of surviving cells in the current turn. When the main Goroutine receives the key "q", it closes the Ticker through “done”.



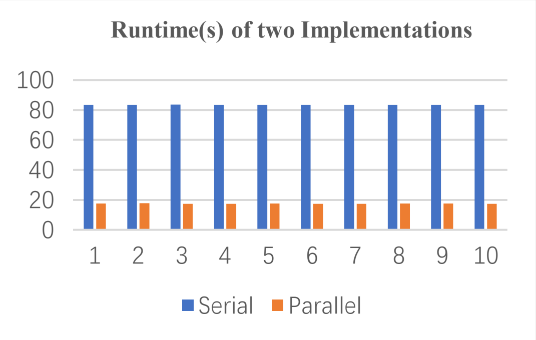
2. Problems solved

When implementing multi-goroutine coordination, shared variables such as world and turn are accessed by each individual goroutine. This concurrent access can result in data races, potentially leading to program logic errors. To prevent such issues, we employ sync.Mutex to safeguard these shared resources. Within each goroutine, we utilize “mu.Lock()” and “mu.Unlock()” to guarantee exclusive access to the shared data. For instance, when the ticker goroutine executes, it first calls “mu.Lock()” to lock the world and turn variables before using them, thereby blocking other goroutines from accessing these variables simultaneously. After completing its operations, the goroutine invokes mu.Unlock() to release the lock and make the shared data available for others.

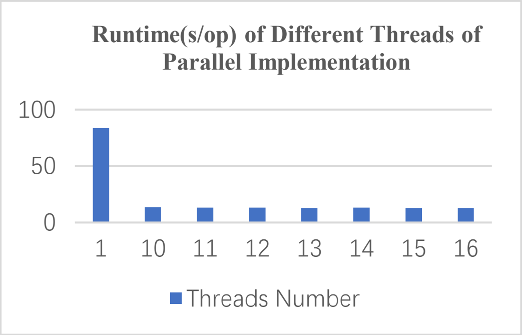
When slices were evenly allocated to each thread, a new issue arose: the boundary cells within each thread's assigned slice depended on the states of neighboring slices' boundary cells to update their own states, complicating the initially balanced distribution. To handle this, we used a duplicate version of the world, called “worldCopy”. As the distributor launches each Worker goroutine, it provides them “worldCopyForWorkers” that includes data from adjacent edge cells. This enables workers to independently compute updates for their edge cells, effectively re-isolating the tasks among workers and restoring efficient parallel processing.

3. Testing and Critical Analysis

We conducted all tests 10 times in a row on the same lab Linux machine with 20 physical cores and 28 threads. During the testing process, we closed other background applications to reduce environmental interference and ensure more stable test results for each run. Our code can fully pass all test instructions. The following are the result diagrams of our benchmark tests.



We focused on analyzing the performance aspect through benchmark tests. Go. The ns/op (nanoseconds per operation) was the core metric, representing the time taken for a single operation to complete 512×512-sized, 1000-turns iterations of the Game of Life. The smaller the value, the better the performance. When the number of threads was 1 (i.e., performing serial computation), the average time for serial implementation over 10 tests was 83.5 seconds. As the number of threads increased and parallel computation began, the mean time of different threads numbers was decreased significantly. When the number of threads reached 13, the average time was 12.67 seconds, with a performance improvement of approximately 83.5%. After that, the performance improvement became smaller and tended to stabilize. When the number of threads increased to 15, the average time dropped to the lowest, reaching 12.6 seconds, with a performance improvement of 85%. The test results show that when implementing the Game of Life in parallel, the multi-goroutine collaborative working and task splitting model can utilize the multi-core resources of the CPU to work, significantly improving performance.



Average time (s/op) over 1-16 Threads over 10 tests

3.1 How our implementation scales when more workers are added?

In order to evaluate performance improvements of Gol due to parallelisation, we used Go benchmarks and adjusted the number of worker thread(1-16) to measure the total runtime of different thread numbers used in the case of 512*512 and 1000 turns. Each measurement was repeated 10 times to make results more stable. The testing equipment is a lab Linux machine, which includes 20 physical cores and 28 threads. To reduce potential interference, each test utilizing a different number of threads will be run separately.

As the number of workers increased, the runtime decreased. With 16 worker threads, the Gol was 84% faster than the serial implementation with 1 worker. The perfect improvement would be a runtime that halves as the number of threads is doubled. This is not the case here, because in Gol the part of I/O was executed serially. The runtime of this part will not decrease when worker threads increase.

The improvements also start to diminish, with the improvement from 10 to 16 workers being much smaller than from 1 to 10 workers. Code doesn't scale infinitely and since the machine's CPU has 20 physical cores, using any more than 13 worker threads which is close this number to offers virtually no improvement.

3.2 Analysis of Constraints in Performance Improvement

Our parallel implementation of Gol can significantly enhance the game's running performance, but it is limited by the serial computing in the I/O part and the number of hardware cores. There is a clear boundary for performance improvement. After reaching 10 threads, although the running time continues to decrease, the reduction rate drops significantly.

This benchmark test achieves automated data collection through the bench_test.go file. During the data collection process, we reduce environmental interference by closing redundant tasks in the background. In the code, we close the redundant IO output through “os.Stdout = nil“ to avoid interference from other outputs and system resource race with the accuracy of the performance test. Then use the "BenchmarkStudentVersion" function Through “ go test -run a$ -bench BenchmarkStudentVersion/512x512x1000-1 -timeout 100s -cpuprofile cpu.prof ” command starts the test, automatically runs all tests of threads 1 to 16, and outputs the results. The code configures the test scenarios through structural parameterization. The number of iteration turns is fixed by "benchLength=1000" ,the image size is fixed by “ImageWidth: 512, ImageHeight: 512”,the number of threads is fixed by for thread :=1; threads<=16 are generated in a loop. Test case names are generated as name := fmt.Sprintf("%dx%dx%d-%d", p.magewidth, p.mageheight, p.urls, p.reads). In the test results, each parameter can be seen more clearly and directly.

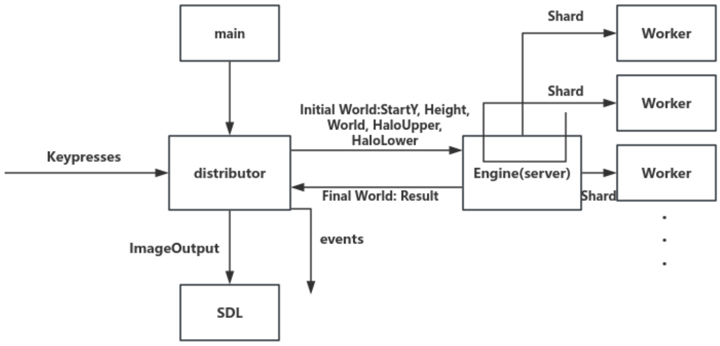
Distributed Implementation

1.The Design of Functionality

According to the requirements for distributed Implementation of GOL, we developed an understanding of how to transition from a parallel to a distributed architecture. This involved transforming the original single-machine, multi-goroutine parallel processing model into a multi-machine collaborative system. Learning from the previously studied Secret Strings problem, we first established an initial system design: the initial separate machine was redefined as a client which meanwhile is a local controller, while AWS nodes were designated as remote servers. Subsequently, we decomposed the original worker tasks from the single machine and distributed them across the AWS nodes, establishing RPC communication channels between the local controller and the server nodes to enable coordinated operation. We created stubs and server, and used the original distributor as the client.

Starting from the serial implementation in a single-threaded, single-machine state, we removed the “worker” and “step” function in distributor.go of the parallel implementation.Then defining the DoWork{} method in server which store the temporary world state by “tempWorld” which includes the data of shard and Halo. Then we created “next” array and transferred the functionality of calculating the next turn of GOL state originally in the local worker function to server. We plan to deploy the Remote Workers on AWS nodes and connected them to the local controller through RPC. We defined WorkerRequest as the RPC structure to carry the game intial parameters sent by the local controller to the nodes, and then defined the WorkerResponse structure corresponding to WorkRequest to hold the calculation results, returning the calculated area status through Result[][]byte. We also define the two structs in stubs and create a method DoWorkerHandler which can be called by distributor through client.call(). The local controller, as the coordination center of the project, contains the core codes such as distributor.go, io.go, gol.go in the Gol directory and main.go as the entrance. The controller is responsible for functions such as keypresses control, starting an RPC call to the remote worker, collecting the computing result of server and merging into global world current, and event notifications. This approach adheres to the Single Responsibility Principle, dividing the original single-machine task into two parts: the controller is responsible for scheduling, and the Worker focuses on computing. This server will be deployed on an AWS node, breaking through the computing power bottleneck of a single machine.

We use RPC as the communication method between the client and the server. In the server, we create the function “RunServer” to start the RPC server which is also built in main.go. The Distributor will send the data of initial world to the server, and the server returns the results to the local server through responses[]. This is somewhat similar to the communication between the Distributor goroutine and Worker goroutine in the parallel implementation through multiple channels, but in the distributed version, we use RPC, which is convenient for network transmission and enables multi-machine collaboration.



1.1 Work Division

Similar to the parallel implementation where the work is divided among multiple worker goroutines to complete collaboratively, we evenly distribute the work each worker in the GOL engine needs to handle. We divide GOL world evenly by ImageHeight into multiple regions. Every worker only need to solve the calculation of row shard. This simplifies the parameters transmitted from the local controller to the gol engine, only StartY and Height need to be passed to determine the shard range of the Worker. It also avoids uneven load on each node.

1.2 Network Transmission

We transmit the parameters StartY and Height required for the initial shard information, the global parameters ImageWidth and ImageHeight, the data of Halo “HaloUpper” “HaloLower” and the current world state World through the network. In the server, they are defined in the WorkerRequest structure. Before starting each turn of state calculation in the controller, the Gol engine needs to receive this information to determine the calculation range of each Worker and the rules of the game execution. After each node completes the corresponding calculation, the next turn state Result[][]byte of the shard will be returned to the controller through the network, corresponding to the WorkResponse structure in server.go. The controller will aggregate the work results of all nodes to generate the global next turn world state. Additionally, we also transmitted the current round Turns, which is done when the controller's timer is triggered. This is because the controller needs to receive the number of surviving cells in the Worker for event notification. When a key is pressed, the local controller transmits the corresponding "pause/exit/continue" instructions to the Gol engine, which can control the behavior of the remote server.

1.3 System Expansion

For this part, we learned the concepts and usage methods. Despite multiple code modifications and attempts, we still failed to complete the construction of Broker. The following is our understanding of its ideal case when it can be implemented.

A new component, Broker, is added to make the system more complete and efficiency. It can help the Gol engine receive instructions from the local server, manage the status of each node Worker, and allocate slice tasks to idle Workers. Additionally, it can help aggregate the next round states of multiple slices and return them to the local controller.

The number of nodes can also be further expanded. The controller will split the world to more shard, the smaller the shard which each worker need to solve, the better the performance. However, it will not increase infinitely. When the number of workers reaches to more, the performance will not significantly change.

2. Problems Solved

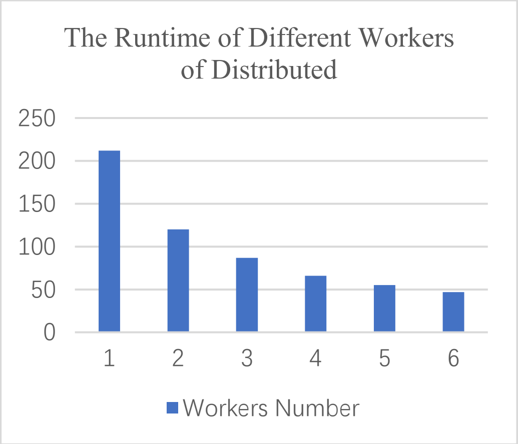
At the beginning, although we created the server, stubs and registered the RPC service in the server, we failed to call it in the distributor. At that time we could still run the entire project through the step function retained in the parallel formula. Later we found that this was only done on a local single machine and failed to achieve multi-machine node operation. Therefore, in the distributor, we built StartDistributed{} as the core for our distributed implementation. Then we created req := server.WorkerRequest, which contains all the data required by the workers, as the target of the RPC request. After that, we connected the workers in the controller and invoked them through methods such as rpc.Dial client.call. Meanwhile we removed the part of function step from distributor.

We know the workers’ tasks should be divided as different but same size shard. But for the edge of every world shard, these cells located at the upper and lower edge will be missing neighboring rows from adjacent shard. Then we defined the HaloUpper and HaloLower in server and distributor. And for HaloUpper and HaloLower. Before the computing of each turn,They will be sent with other initial shard data from controller to workers, which temporarily as the part of tempWorld.

3.Testing and Critical Analysis

We ran the test command "go test ./tests -v -run TestGol/-1$", the test command of step 2 and benchmark tests on a 20 -core 28-thread Linux machine in the laboratory. Each test was conducted five times on the same machine to reduce errors and randomness.

We verified the correctness of the single-machine single-Worker splitting logic in step 1 through the test command. The tests of step 1 and 2 passed smoothly, and we then conducted benchmark tests. In benchmark test, we add codes “workerAddrs := []string{}” to define the address which we would test. By comparing the results of this benchmark test with those of the parallel implementation part, we found that the running time of the tests significantly increased. This was due to the time loss caused by calling remote Workers. Analyzing the results of this benchmark test, we found that as the number of Workers increased, the running time for completing the Life Game significantly decreased. However, as the number of Workers continued to increase, the improvement in running time became smaller and eventually stabilized.



4. Potential Fault

When the network connection is interrupted, the communication between the Controller and Workers is disconnected, and the RPC requests or responses will fail. The local controller's instructions cannot be transmitted to the server, or the server's calculation results cannot be returned to the controller, and the system will be unable to continue calculating and aggregating the next round's state.

When some Workers experience anomalies, the local server will lack the states of certain shards, resulting in errors in the aggregation of the returned state results from the nodes and the inability to generate the next round's global GOLstate.

If the local server suddenly crashes during the execution of a certain round, the world state at the current round cannot be saved, and after recovery, it may be necessary to start from the initial state of the world.