Casper Wang yf20630
Mingzhang Deng wu20880

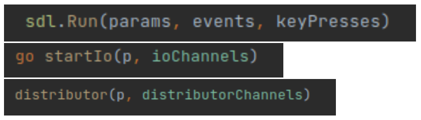
Stage 1 - Parallel Implementation

1. Functionality and Design

1.1 System and Functionality

For the parallel implementation, our team has completed all tasks shown in the guide and added extra optimization and functionality to it. Starting from the beginning, the implementation is made up of 3 major components: the distributor, IO and SDL. All components within this framework run concurrently. Though they might not be communicating all the time, at some moment in time, they will be doing channel communication to send or receive information from one another. This way, the whole system can work efficiently as it utilizes the multi-threaded ability that modern computers can offer us.

To be more specific. Firstly, all three components will be launched almost at the same time, this is shown in Figure 1.



```
sdl.Run(params, events, keyPresses)
go startIo(p, ioChannels)
distributor(p, distributorChannels)
```

Figure 1: Code snippet of the goroutines

Because of how each component contains loops within, the three components will be running concurrently until either the input has been done or the user decides to terminate it. Within the distributor.go file, there are three sub-components (goroutines) that will be attached and interact with the distributor component. Those three components will be running in a concurrent fashion, and they each correspond to a functionality they bring to the system: 1. Tickers, which reports every two seconds the completed turns and the current alive cell counts to the IO component through the IO channel 2. Keypresses, which relates to the behaviour the SDL should react as when the user presses a certain key on the keyboard. 3. Multithreading, which gives the system the ability to spread the computation of turns out of multiple threads.

1.2 Problem solved

The major issues that may arise with a system like this would be the coupling and the potential race conditions between those goroutines. Because they are all located within the same package, ultimately this means they will have some sort of shared memory, in our case a struct as a pointer that got passed around in the distributor. We introduced a mutex lock and a semaphore to the system, this way we could circumvent any potential race conditions.

Another interesting problem that we solved is about how to split up the work efficiently to multiple threads. The tests require the program to support any number of threads from 1 to 16, in which how you implement the odd number of threads become a problem. As you simply can't divide a power of 2 by an odd number without getting remainders. We did come up with a naïve solution where we assigned an equal amount of work to the first n – 1 worker but left the rest to the last worker. But this is not ideal. So we optimized by first multiplying the index then dividing the thread number, this way each worker will be assigned an almost equal amount of work and the height won't be differed by 1.

2. Critical Analysis and Testing

2.1 Benchmarking

All benchmarking results were attained from running on a 6 core 12 threads Linux lab machine within the same day. We made sure that there are as little applications running on the background as possible during testing to achieve the best performance.

All tests were run on a 512X512 world for 500 turns, this process will then be repeated for 5 more times and be averaged to get a more precise indication.

2.2 Serial vs Parallel

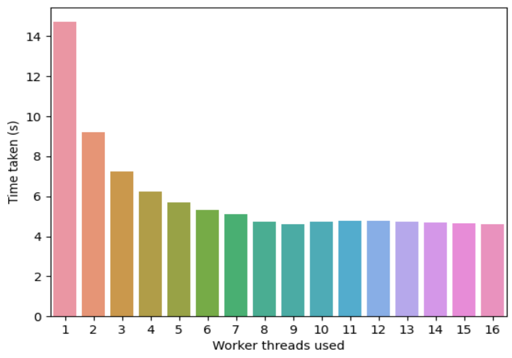


Figure 2: Average time per operation (512x512x500) over a range of threads over 6 trials

It’s quite easy to see what parallelisation can do with our system, the performance boost is massive, using 16 thread is 3.2x faster than the serial version. This performance boost can also be seen from the very start. Even with one more addition of the thread, the performance is already 60% faster than with a single thread, it proves that paralleisation does work, and it can be very beneficial when it comes to large systems.

2.3 The performance analysis

Figure 2 gives an almost exponentially decay graph, the reason why is almost will be discussed later. But the overall principle is that the more worker threads added to the system the faster the runtime. We can also see that there is a diminishing return when it comes to adding more worker threads to the system. The ideal scenario would be the performance doubles when the number of worker threads doubles. This is not the case here, as the system consists of multiple components, but only the worker threads are optimized with parallelisation i.e. having more threads working on those tasks. The rest, for instance, getting the initial world out from the channel or generating PGM and sending it down to the io channel byte by byte, are not parallelizable. Therefore they won’t benefit from the addition of worker threads, and the runtime would more likely be exponentially decayed instead of a linear function.

2.3.2 What’s with that weird shape?

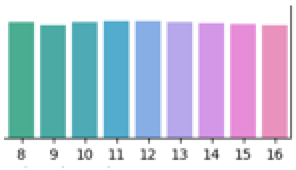


Figure 3: Snippet of the Benchmark results ranging from 8 to 16 worker threads.

As I have previously mentioned, there is a slight increase in runtime when adding more worker threads e.g. from 9 to 10. Also that there isn't much performance difference from 8 worker threads onwards.

One thing to note is that the lab machine has an i7 8700 6 cores CPU, which has the ability of hyperthreading to simulate as 12 logical threads.

One reason I could come up with is that, though the CPU has 12 logical threads in total, obviously there are things running on the background that might take up a few threads, not to mention that the IO and SDL could potentially also be using threads too, so in theory, the performance gain would end at some point, in our case, it could possibly be at thread 9. After this, when we are adding more goroutines to the system, the go runtime would receive more workloads, but the system is already at its limit point, so I would assume it might take extra time to wait for other threads to finish so the leftover one can proceed, therefore an increase of the time taken. But as more goroutines are added, the go runtime might have some way to optimize and allocate better the workload it has, thus a minimal performance gain again.

2.4 Optimisation

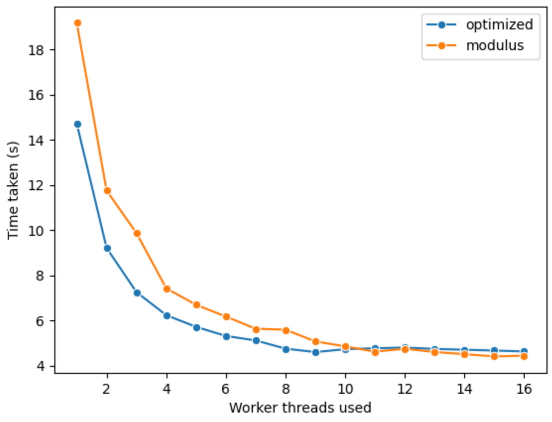


Figure 4: Comparison between the two methods.

Having known that the modulus operator in Go takes a long time to compute, we decided to optimize our GOL logic by using plain for loop and if-else statement instead of modular operations.

The performance difference is quite substantial where the optimized version is 26% faster than the single thread and 23% faster than the modulus version with double threads.

I have absolutely no idea why the modulus version would overtake the optimized one from 10 threads onwards, it could be accounted for many many reasons. Perhaps the CPU went a bit grumpy and decide to overclock itself so it can finish the damn work quicker and take a break.

2.5 Potential further optimization

We observed that a few processes can be concatenated together to achieve faster runtime. For instance, the CellFlipped event can be sent along with the process of the GOL logic, because both of which needs to go through the entire 2-D world to do their operation, and this would save a lot of computation when the turns are getting bigger, I would assume that there will be a significant difference. To not break the integrity of the implementation, we decided to not make these changes as if so we will have to have workers to deal with the SDL threads which would increase the coupling and is not what we desired.

This improvement can be seen from the other parallel version that we shared in one drive.

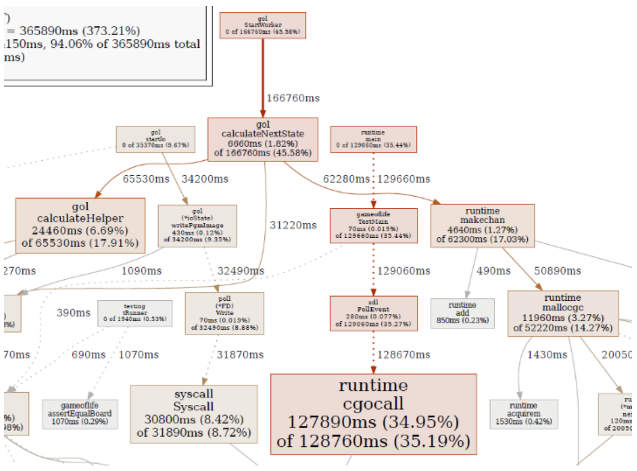

gol calculateNextState 6660ms (1.82%) of 166760ms (45.58%)
gol calculateHelper 24460ms (6.69%) of 65530ms (17.91%)
runtime makechan 4640ms (1.27%) of 62300ms (17.03%)
runtime mallocgc 11960ms (3.27%) of 52220ms (14.27%)
syscall Syscall 30800ms (8.42%) of 31890ms (8.72%)
runtime cgocall 127890ms (34.95%) of 128760ms (35.19%)
166760ms
129660ms
129060ms
128670ms

Figure 5: partial pprof result of the implementation

Also, from profiling our parallel implementation using pprof, shown in Figure 5 we noticed that runtime cgocall takes the longest time. We do not know exactly what cgocall does, but it's referred to as runtime, so all we can assume is that it has something to do with go runtime. In our implementation, we passed the entire 2-D world every time the worker goroutine is called. And this process is repeated Turns * Threads time which could easily be hundreds and thousands of times. It could be a big workload!

According to those, we made an educated guess that those two might be related. So instead of passing the entire world but only the part each worker needs to process as well as with the halo region, this load could be diminished a lot. Due to the time limit, we couldn’t verify our guess, but just sending halo could potentially be an improvement to our system.

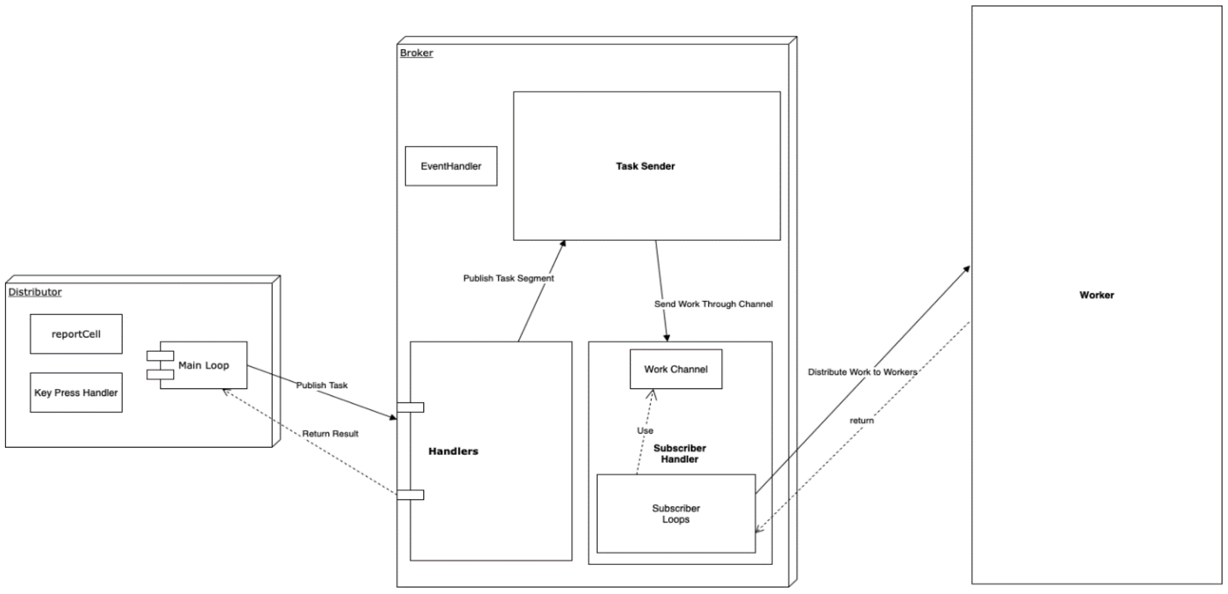


Figure 1: Distributed System Design in one Diagram

Stage 2 - Distributed Implementation

We separate the entire system into three different levels of abstraction. At the system level of abstraction, we will introduce how systems interact with each other to accomplish a simple task. When comes to the Component level of abstraction, we will describe how component achieves their responsibility and their workflow. At the lowest level of abstraction, we will introduce some detailed implementations in critical sections of code.

1. Functionality and Implementation

For the distributed part of Gol, we separate it into three different parts: distributor, Broker, and Workers. For each part, they take part in their responsibilities with their workflow.

The distributor is a User interface, It mainly takes part on interact with users and changing the system state by user's operation. We put every function that can be calculated locally into a distributor (Works that are not related to Gol logic). The distributor can control distributed gol system by sending a different request to the server.

Broker is a redirect centre of works. All works from the distributor are sent here. The broker takes responsibility to communicate between distributors and workers.

Worker is a server that can calculate Gol in the designated part of the Gol world. It takes \ responsibility to calculate the result and send back the calculation result through response.
In our implementation, workers are only required to calculate one turn for each request.

The specialty of our implementation is we can dynamically add and shut down our workers. Even if there’s no worker available, the entire system will not shut down. Broker will save current state waiting for another worker to subscribe it.

To achieve fault tolerance, we lend the idea of tuple space as our fundamental methodology with a delicate take&read access control. Every component in this distributed system is able to publish/take out the task to/from the work pool. However, unless the receiver successfully gets the result and sends back to the publisher, the subscriber loop will put the work back into the work pool. This mechanism makes sure that 1. Only one worker will handle one task.
2. If worker failed to send respond, the work will not disappear. Therefore, if a worker shutdown when working on a task, the task will not disappear and will be arranged to another worker.

2. Component Detail

We separate distributor into three main parts: IO routine, Event Handling routine, and a distributed system controller routine. IO routine do IO operation according to the Event and info send through ioCommand and IoInput/IoOutput channel from other parts of distributor. Those can be seen in Figure 1.

Event Handling routine can handle the keypresses event from the user and interferes the main working loop or start other supporting routines to achieve function. In distributed system controller routine, you can publish your task or control command to the broker through stubs defined in stubs.go. We achieve functions below:

1. Publish task to broker.
2. Request the current calculating process and get the world and correlated completed turn.
3. Stop distributed system.

We split broker into four parts: Task handler, Subscribe handler, Work pool, and Event handler. Their relationship is shown in Figure 2

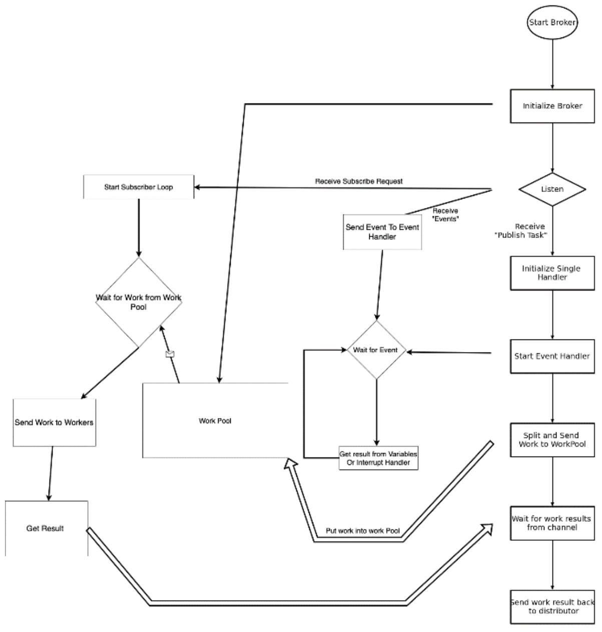

Start Broker
Initialize Broker
Listen
Receive Subscribe Request
Start Subscriber Loop
Receive "Events"
Send Event To Event Handler
Receive "Publish Task"
Initialize Single Handler
Wait for Work from Work Pool
Wait for Event
Start Event Handler
Send Work to Workers
Work Pool
Get result from Variables Or Interrupt Handler
Split and Send Work to WorkPool
Get Result
Put work into work Pool
Wait for work results from channel
Send work result back to distributor

Figure 2: Broker internal workflow

In broker, task handlers mainly communicate to distributors, and subscriber handler mainly deals with workers. Before starting work, the worker will send subscribe request to Broker, then the broker will create a subscriber loop for this request and wait for publishing tasks to the worker when tasks are available. The broker will receive work detail and a specific ID from a distributor. Then it will call functions to split task into task segments and put task segments into a work pool. A collector will collect tasks and send them to a Subscriber loop. The subscriber loop will send received work to the designated worker. When the subscriber loop gets the work result, it will send the result back to the specific handler according to the ID. At last, the handler will gather the task and send it back to the distributor.

The worker receives requests from a Broker and handles the task. The worker works basically the same as the worker in the parallel part with an RPC entry. We will not say more on this part.

3. Benchmark result.

We test the performance of the GOL with different numbers of workers online.

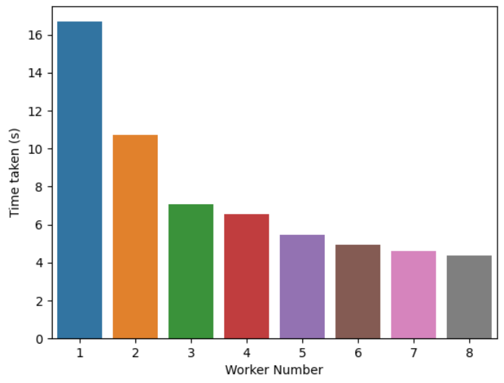


Figure 3: Time spent on calculating 512x512x100 Gol(Local)

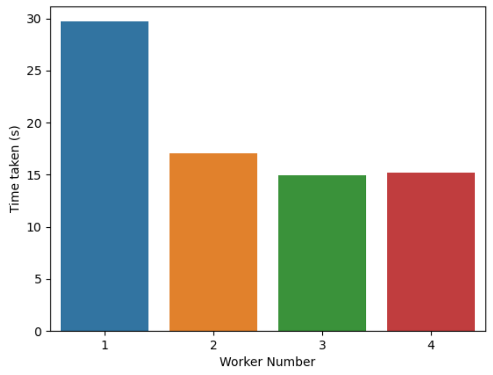


Figure 4: Time spent on calculating 512x512x100 Gol(AWS)

As we can see from the graph. Increasing the number of workers does reduce the time spend. The trend is obvious on the local test. When comes to the AWS test, we can see the curve flattened earlier. This is because we do speed up on calculation, but we increase the portion of communication cost at the same time.

After this, we test how efficiency changes when we let workers run in parallel.

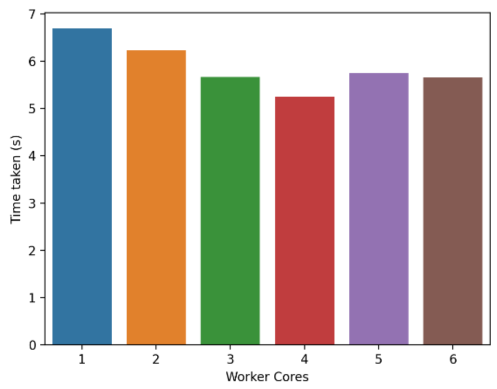


Figure 5: Gol 512x512x100 in 4 worker servers.

4. System design and Decision making

Our priority on designing distributed system is:

1.Reliability
2.Extendability /Replaceability
3.Efficiency
4.Easy to use

In distributed part, we have several different patterns that can use.

One big decision we make is about which parameters should we pass across the different machines. At last, we analyze the most important info that must be transmitted between machines.

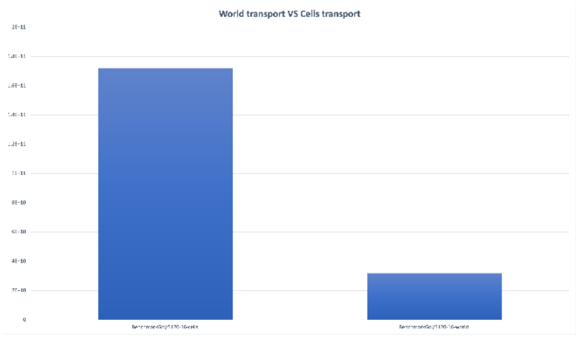


Figure 6: World transmission vs Cells transimission

Here we tested the time difference between transporting the entire map and creating a new map after getting a list of alive cells. Shown in Figure 6

We thought that sending Cells between machines will be a better choice. However, the test result shows that the time we used in dealing with cells use even more time than transporting them. Therefore we choose to send the entire world between machines.

Final Conclusion

From all those explorations, we are absolutely amazed by what modern technology is able to deliver nowadays. Let me remind you again the 3.2x performance boost from fully utilizing all 12 threads of the CPU, and the 2x gain when distributing work to 4 AWS nodes. And this is not even the best we can do, with the technology constantly evolving. I am sure that commercial laptop that has CPU with 12 cores, 20 cores would soon come to life. Running our tests on multiple machines like this fully utilizing all its multithreading power would sure be another story. Clearly, Concurrency and Distributed System is the trend and the computation power it could bring would benefit our life largely in the future.