Game of Life Coursework Report

Group Members: Zik Zhao (gp23281@bristol.ac.uk),
Zixuan Zhu (kh23199@bristol.ac.uk), Chenxing Liu (ya23880@bristol.ac.uk)

Part I: Parallel Implementation

1. Functionality

Our parallel implementation supports all the features required in a highly optimised way:

- Fast evaluation of GOL logic using number of threads provided, using shared memory where a condition variable and a channel are used as the synchronisation primitives between distributor and workers.
- Initiating IO request and obtaining results later to read and write cell matrix from and to physical files
- Having a live view of the progress of game.
- Sending alive events every two seconds.

2. Final Design

Distributor, logic workers and block division function are defined in file *distributor.go*. Matrix structure and matrix manipulation logics are defined in *matrix.go*.

We use block division algorithm to allocate tasks to each worker. This algorithm tries to make rectangular blocks evenly sized, and close to squares instead of long strips. In each block, there are safe region and unsafe boundary. A cell is in safe region if all its neighbouring cells are in the same block, otherwise in unsafe boundary.

Evaluation steps:

1) Send an IO request and obtain pixel data to construct a matrix structure. Another empty matrix structure is created as well for storing results for this turn.
2) Divide matrix into multiple blocks depending on number of threads.
3) Create goroutines, providing parameters, matrixes and synchronisation primitives.
4) In the main loop, broadcast signals to awake all worker routines so they will evaluate a single turn, count change and a slice of flipping cells in unsafe boundaries are sent back via a channel called `result_chan`.
5) The distributor waits until all results are sent back, then the distributor updates count of alive cells and counts of surrounding cells for these cells in unsafe boundaries.
6) After that, the distributor handles keyboard events. Loop is broken for quit command, `write` function is called for save command. For pause command, the pause flag is set, and control flow is restricted to the event handling loop. Relevant events are sent through channels.
7) After each turn, the references to source and target matrixes are swapped.
8) When all turns completed or interrupted, running flag is unset and all worker routines are signalled so they will exit. Final matrix is saved, and events are sent as required.

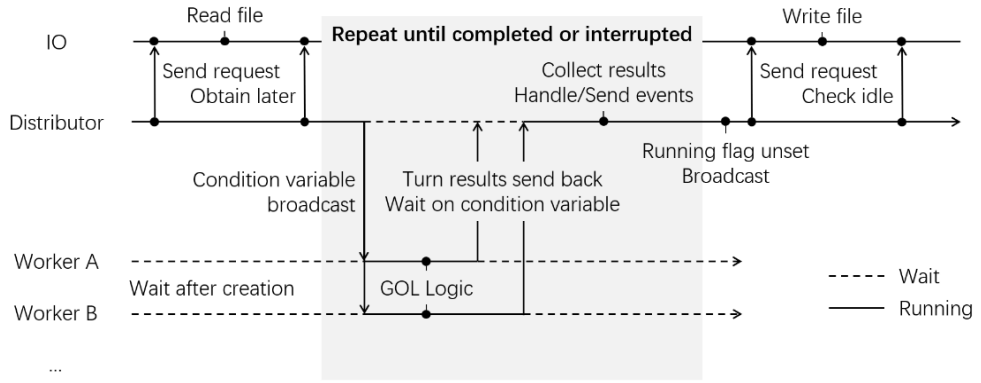

Figure 1: Diagram showing how routines interact with each other

3. **Advantages**
   1) Using only a condition variable and a channel for synchronisations between workers.
   2) Storing and maintaining counts of surrounding alive cells reduces workload since number of cells flipped each turn is far smaller than total number of cells in the matrix.
   3) Optimised IO functions that splitting IO requests into two steps. Distributor can initiate an IO request and gather results later.
   4) Workers are reused for the entire task. No overhead for creating goroutines again.
   5) Only two matrixes are constructed, and their references are used directly by workers, instead of copy of blocks in the matrix.
   6) Buffers for storing cells flipped each turn is reused throughout the task. Cells flipped are copied and sent to SDL component as slices.

4. **Disadvantages**
   1) Higher consumption of memory for storing counts of surrounding cells for each cell.
   2) Only composite number of threads are supported due to limitation of block division algorithm. Prime number of threads are floored to nearest composite number.

5. **Test Results and Benchmarks**

   All tests passed with race detector enabled.

   All benchmark tests are running in the same lab machine in one session to control variables. All benchmark tests are run 10 times and abnormal results are ignored and retested.

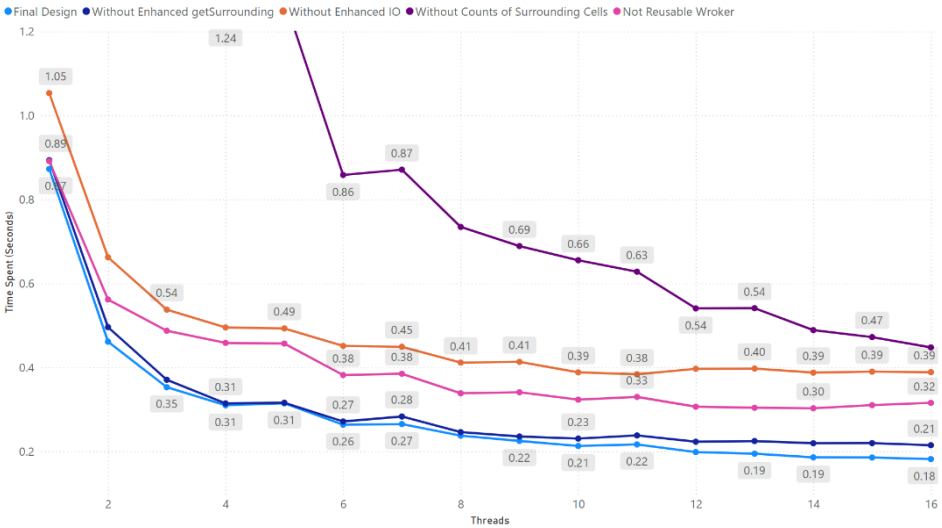

Figure 2: Time spent for 512x512x1000 with variable number of threads in different implementations

From this graph, we can conclude that the introduction of counts of surrounding counts remarkably enhances the performance of evaluation. It takes about 4 seconds for single thread without that improvement but now it only takes about 0.8 seconds, which means workload is reduced to about 1/4. However, this difference is shrinking when they have more threads due to increased number of cells in unsafe boundaries, and, therefore, leads to more work for the distributor.

Apart from that, refined IO also leads to significant improvements. In previous implementations, IO component interacts with the distributor by sending each byte separately through multiple channels, which means synchronisation takes place for each byte. That is a disaster for the program.

Reusable worker goroutines save time spent on creating goroutines and garbage collection.

Finally, latest changes to the `getSurrounding` function which gets the positions of neighbouring cells brings a slight improvement on performance. Its performance curve is close to that of final Implementation. CPU profiles for these two implementations provide evidence for this

improvement that CPU usage for that function decreased from 6.33% to 0.76%.

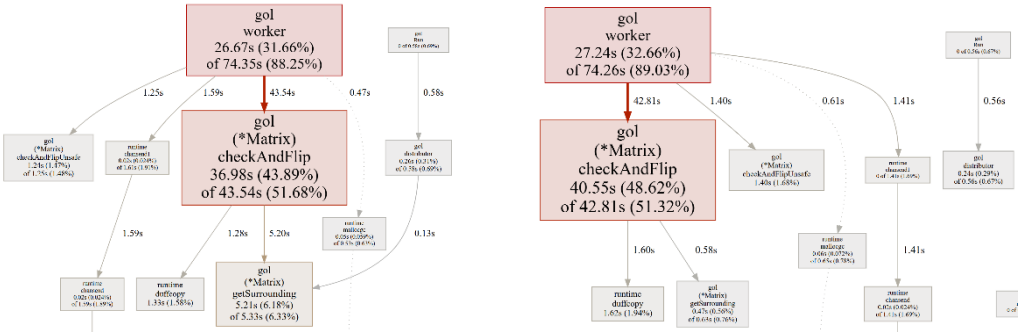

Figure 3: CPU profiles for implementations using original(left) and enhanced(right) `getSurrounding` function (benchmarking on same task)

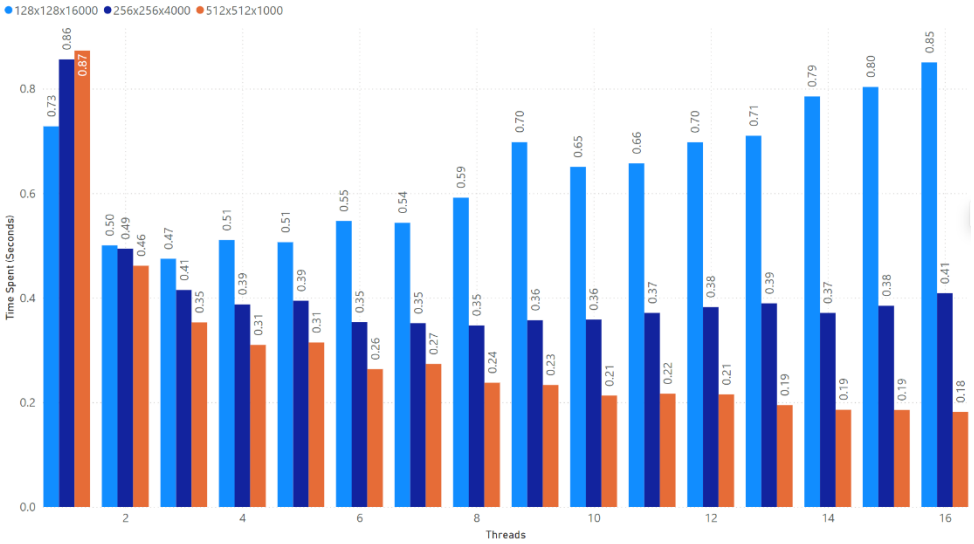

Figure 4: Time spent for three different tasks with variable number of threads in final design

This graph shows three distinct trends that the program reacts with three different kinds of tasks, where those tasks have equivalent number of computations. All of them experienced a significant decline in evaluation time when they get two threads. But then, time spent for the task with smallest cell matrix starts increasing as number of threats increases. The medium one increases too in a gently rate. We only see a decreasing trend in time spent in larger matrix, but it is becoming stable as thread number rises. This graph shows that our program is good at evaluating larger cell matrixes since time spent for unsafe boundaries are not significant for larger matrixes.

Part II: Distributed Implementation

1. **Functionality**

   We started working on this part after our parallel part is fully implemented and tested, so our distributed implementation is based on our parallel design. It supports all the functionality listed in parallel part, including SDL live view.

2. **Final Design**

   The cloud computing system consists of one broker and multiple worker nodes. There are logic workers in each worker node which is truly the part doing GOL logic. This system is accessible via RPC dialling to broker's static IP address. Task can be evaluated by establishing TCP connection to broker, making RPC request to broker's `Init` function, which dispatches matrix and assign subtasks to worker nodes. The broker then starts a loop to repeatedly instruct worker nodes to evaluate next turn and collects results until completed or interrupted. Cells flipping events from worker nodes are redirected to local controllers via said TCP connection.

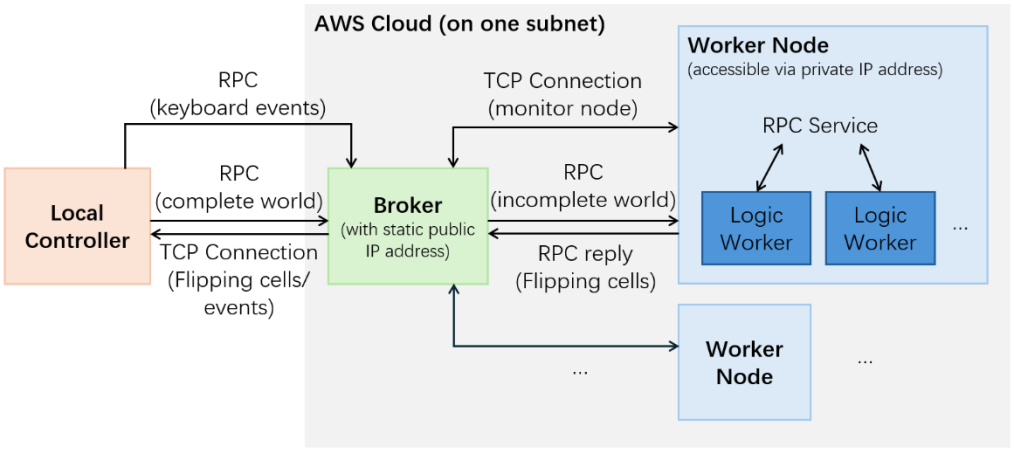

Figure 5: Architecture diagram of distributed part

3. Special Design

1) Compression

All complete matrixes and cells flipping events are compressed before transmission to save bandwidth. Matrixes are compressed so that each byte stores values for 8 pixels. Cells flipping events are compressed that each integer only occupies the minimum number of bytes to represent the whole range of width and height.

2) Exchange graph

As each partition is assigned to a worker node, the exchange graph saves exchange targets for each cell. When the broker collects turn results from worker nodes, it produces a slice of cells flipped for each worker node, where they are applied to incomplete matrix in each node before evaluation of next turn. Those applied to matrixes are called adjustments.

3) Scalable system

The IP addresses of worker nodes are not hardcoded in broker code. However, each worker node has private IP address of the broker, and when they start running, they try to establish a TCP connection with broker and the broker can get their private IP addresses from the connection to utilise their RPC service.

4) Multiple logic workers in each worker node

Partition is a set of blocks that assigned to each worker node. In each worker node, multiple logic worker routines are created to evaluate each block. They act like what we have in parallel parts. The thread number of task is the total number of logic workers in nodes.

5) Receiving data in a stream

Since cells flipped events are needed to be sent back to local controller each turn to make the SDL live view working, we established a TCP connection between the local controller and the broker. Results for each turn are driven back via this connection to local controller and we have event type enumerations to help local controller distinguish different events. Length information is also added at the front of each message to prevent sticky packets.

6) High fault tolerance

The system is working unless either broker is offline or no worker is connected. The following errors are handled:

- When unexpected connection failure between local controller and broker occurs, local controller is panic but broker outputs an error information and continue working.
- When unexpected connection failure between broker and worker occurs, broker removes that node from available node list, disconnected worker node tries to reconnect to broker once per second until it's reconnected.
- When unexpected connection failure between broker and worker occurs that is running a task, the broker calls `recover_evaluation` function that restarts the task and the task continue evaluations from saved turns. This function uses existing

connection to local controller but reassigns works for remaining worker nodes. Thus, there is no requirements on the boot order of this system.

4. Advantages

a) Log is widely used to monitor the whole system.

b) Keyboard events are sent to broker, when broker handled these events it sends the same events back to local controller to synchronise status of local controller and broker. Otherwise, the broker still processes a few of turns due to delay of receiving events.

c) Both local controller and broker keeps a copy of the entire cell matrix, so the local controller only needs a slice of flipping cells to update matrix, and broker's copy helps recovery of evaluation when any worker is getting offline. Each worker keeps an incomplete copy of cell matrix that only contains data in assigned partition.

d) Logic worker goroutines in worker nodes are reused for all the turns.

e) All cloud machines are launched at the same region so they can connect to each other via private IP addresses. Only broker is assigned with an elastic (static) IP address.

Some other advantages are already explained in special design section.

5. Disadvantages

f) More memory consumed to store counts of surrounding alive cells, exchange graph and worker node information.

g) TCP connections are not reusable. A new connection is established for each task. This helps preventing unread events for last tasks being read by current task if current task is executed before routines for last tasks completely exit. New TCP connections may be relatively slow.

h) Sending events to broker instead of reacting locally causes a delay in response.

i) Maximum worker nodes available is 8 since we use byte as the type of elements in exchange graph. Changing to larger data types increases capacity but consumes more space.

j) Singleton. The system can only run one task at a time, other incoming tasks will be blocked as they cannot acquire mutex.

6. Test Results and Benchmarks

All tests passed with race detector enabled.

All benchmarks are tested locally via Microsoft Hyper-V virtual machines instead of cloud ones since we don't have permissions to launch instances with preferred configurations.
The following graph shows performance of this system as number of threads increasing. Benchmark is done in broker (Local controller is not suitable for benchmarking since network is a large influence factor which cannot be avoid). The system consists of one broker and two workers, each worker has allocated 8 virtual cores. Memory is set to 1GiB for each one, but dynamic memory is enabled.

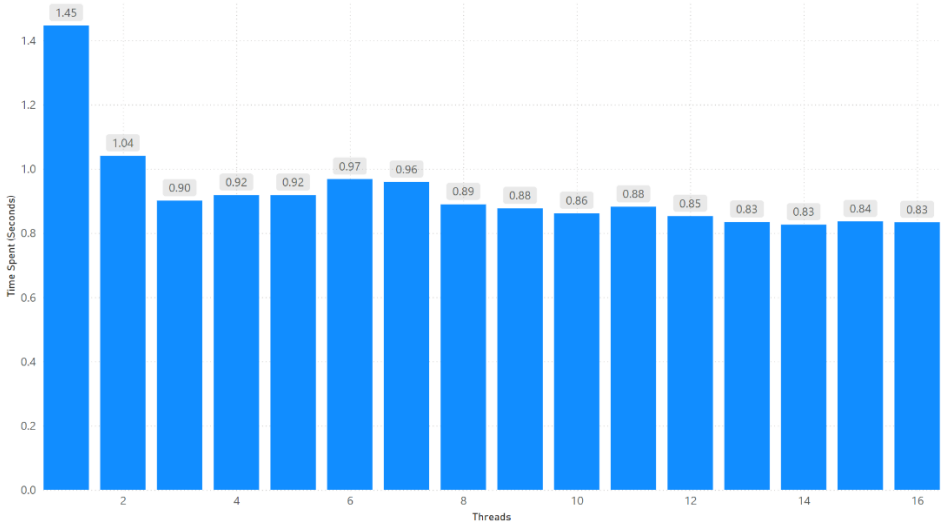

Figure 6: Time spent for 512x512x1000 with variable number of threads

The following graph shows time distribution for the same task, but we are using one broker and one worker with 16 virtual cores. It is the results of two benchmarks, one running in broker and one in worker, and the dark blue bar is their difference, which is the total time spent one communication between broker and workers and works for broker. The mean of difference is 0.332 and standard variance is 0.019, then we can assume that difference is a constant.

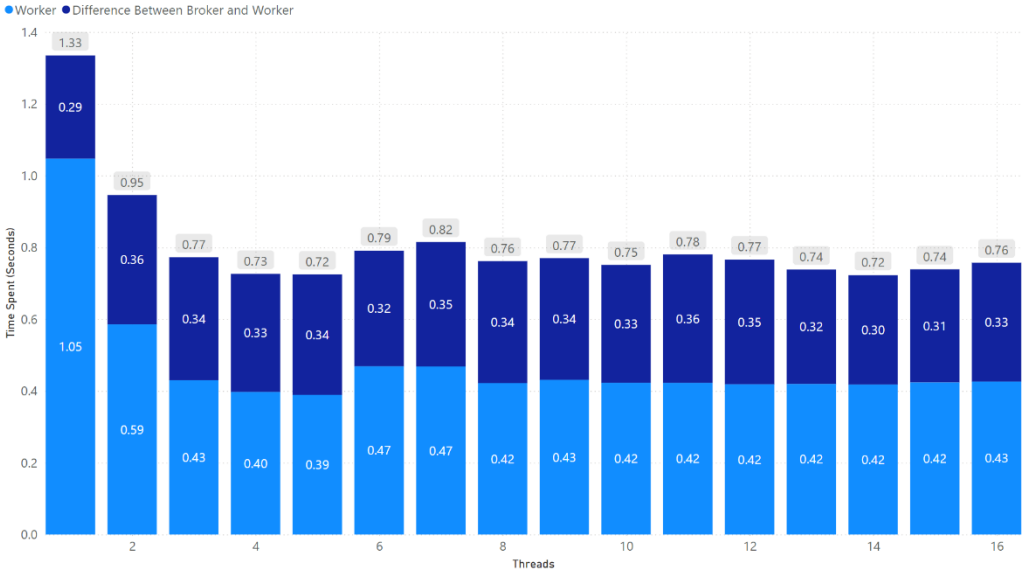

Figure 7: Time distribution for 512x512x1000 with variable number of threads

7. Potential Improvements

The broker acts like a load balancer that incoming tasks are divided into as more partitions as possible, which means when the system is hosted in powerful cloud machines, the load for each node is quite comfortable as it is singleton and loads are shared. We can improve the system by introducing task queue and load monitor, to allow the system to evaluate multiple tasks at any time while assigning moderate amounts of workload for each node.

Conclusion

This coursework provides a great opportunity to practice our sense of concurrency and synchronisation. We have widely used condition variables instead of mutexes and we realised that condition variable is truly a quite efficient synchronisation primitive that some routines are waiting for the command from the controlling routine, just like the way people interact with each other. The key to concurrency is clear task allocation, with minimised data races. Our strategy is always consuming more spaces for better performance, many designs in our programs reflect that.

We also used CPU profiles to analyse the performance bottlenecks of our system. The graph provides a convenient way to find the most consuming components and the source code page allows us to find CPU usage for each line. With aid of this tool, we have proposed more enhancements like faster calculation of positions of neighbouring cells. After optimisations, we checked the CPU profiles again which showed an increased percentage of CPU time is spent on GOL logic which is what we desired.

For the interaction with SDL component in both parts, we prefer to leave its implementation as channels. Based on our knowledge of handling window events in application programming, message queues and message loops are normally used to make the windows working. We think using channels is a more naturally way here since they are close to message queues, and it helps to reduce coupling of the distributor and SDL components.

All data figures are made in Microsoft Power BI. All diagrams are drawn in Microsoft PowerPoint.