Parallel and Distributed Implementation of Conway's Game of Life

Karlee Sun (lj21475), Frederick Zou (un21170)

November 2022

Stage 1 - Parallel Implementation

1. Functionality and Design

1.1. Implemented Functionalities

The program is designed to simulate Conway's Game of Life (*GoL*). Our implementation begins with a single thread worker that iterates through each cell for every turn and saves its next state in a new 2D slice. We parallelized the programme based on the serial implementation, allowing the user to specify the number of threads to be used. The world was divided into slices of consecutive rows and assigned to workers. Each worker is started as a goroutine and will send back a 2D slice of their sub-world after one turn via the *outChain* channel. The distributor will append the world slices in order.

More user interactions were added to the program. We added a "Select" statement inside the "for loop" for turns, with the game of life logic being the *default* case, and two cases for receiving the signal from *tickerChan.C* channel and the *c.ioKeyPresses* channel. The *tickerChan.C* channel will send a ticker every two seconds notifying the distributor to send the number of alive cells using the *AliveCellsCount* event via the *c.events* channel for it to be printed to the console. In addition, whenever the user pressed a key, it will be sent through the *c.ioKeyPresses* channel to the distributor. The *keyPress* is determined by a "Switch" statement and then executed accordingly.

1.2. Challenges Overcame

In this program, workers access the world using memory sharing; to prevent the possibility of race condition, workers will only get a slice of the world and will store the new world in a new 2D slice. The distributor will append and store it in the *newWorld [][]uint8*. After each turn, the distributor will switch the pointer accordingly.

The world is wrapped around borders, which would be difficult to pass as one slice to workers. As a solution, each worker will not only get the sub-world but also *topEdge []uint8* and *botEdge []uint8*, for the row above their working area and the row below.

The control rule for '*q*' should pause the program and the event *AliveCellsCount* that happens every two seconds should also be pausing. So we introduced a "Select" statement with *GoL* logic being the default case. When '*p*' is received from the *keyPresses* channel, it will start a blocking "for" loop until another '*p*' is received for resuming. The use of busy waiting will be discussed in part *3*.

Initially, every worker will be iterating through every cell, counting its neighbour by calling *countAliveInOneRow()* for the row above the cell, containing the cell, and below the cell; and then calculate its next state (*this implementation will be referred as the "Unoptimized version"*). Whereas the optimized *countNeighbour()* function only iterates through each cell once and stores its alive neighbour in the *neighbourCounts []int* slice (*this implementation will be referred as the "Optimized version"*). More details will be discussed in part *2.2* and *2.3*.

2. Performance and Benchmarking

2.1. Test Environment

	macOS	Windows	Linux*
CPU	M2	Intel i9-9900K	Intel I9-9900K
Core	4 performance 4 efficiency	8	8
Threads	8	16	16
Memory	8GB	64GB	64GB
OS	Ventura 13.0.1	Windows 10	Parrot 5.1
Hyper-threading	No	Yes	Yes
Go Version	1.19.2	1.19.1	1.19.1

Table 1 Test Environment

*Kernel Linux and Windows are operated in the same machine as Dual-boot and were installed on a mechanical hard drive.

Every benchmark test was repeated 5 times from 1 to 16 threads on the same 512x512 initial configuration for a 1000-turn game and was completed after restarting the machine to minimize uncertainties. In addition, tests were run under macOS, Windows, and Linux for cross-comparison and to see the performance with minimal disturbance from the environment. This also allows us to observe optimizations made by the OS.

2.2. Analysis of the Unoptimized Implementation

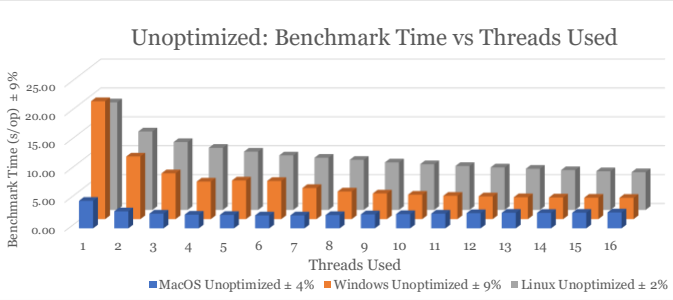

Figure 1 Unoptimized: Benchmark Time vs Threads

Figure 1 shows the result of the scalability test for unoptimized implementation under different environments. There is an inverse correlation between the number of threads and runtime. As threads increased from 1 to 16, the program was 1.7x faster on macOS, 5.5x faster on Windows and 3.0x faster on Linux.

The improvement diminished as the number of threads increased. This might be due to the parts of the program that cannot be parallelized, such as loading the PGM file. The benchmark time generally decreases for Windows and Linux, however, for macOS, the benchmark time increased by 0.062s from 8 threads to 9 threads. This might be due to the hardware limitation, as Windows and Linux allow 16 workers to operate on 16 different physical threads with hyperthreading; but for MacOS, multiple threads would start to use the same physical threads from 8 threads onwards.

Furthermore, the overall benchmark time for MacOS is significantly less than the others, this might be due to the optimization made by the OS for M2 CPU. Moreover, as Windows and Linux are run on the same machine with the same hardware design, the Windows system might also have made some optimizations to the program, so it runs faster than on Linux.

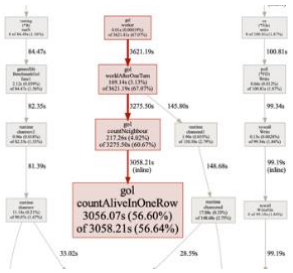

Figure 2 CPU Profile for Windows (Unoptimized)

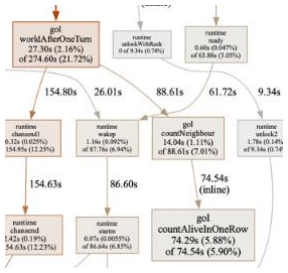

Figure 3 CPU Profile for macOS (Unoptimized)

By producing a CPU profile of count 10 on Windows (*Figure 2*) and macOS (*Figure 3*), the optimization made by M2 CPU itself can be observed by seeing the *gol countAliveInOneRow* function.

The time taken by the *countAliveInOneRow()* called by *countNeighbour()* is occupying more than half of the time of the entire program, so the focus of the optimization will be on improving the method for counting the alive cell in the neighbourhood of cells.

2.3. Analysis of the Optimized Implementation

In the unoptimized parallel implementation, workers iterate through every cell and traverse the value of its neighbours. Whereas in the optimized version, workers iterate through the given world once at the beginning of every turn, and if a cell is alive, add one to its neighbours in *neighbourCounts []int*; it will be used as a lookup table when calculating the next state of a given cell. In practice, the optimized version significantly reduces the benchmark time:

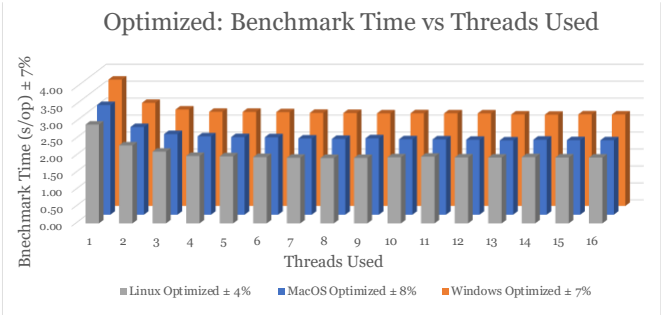

Figure 4 Optimized Benchmark Time vs Threads

Figure 4 shows the benchmark time after optimizing the *countNeighbour()* function. Benchmark times generally follow the same pattern as the

unoptimized version, but there is a smaller time difference between the number of threads. As threads increased from 1 to 16, the benchmark time is faster 2.19x for MacOS, 2.70x for Windows, and 1.94x for Linux.

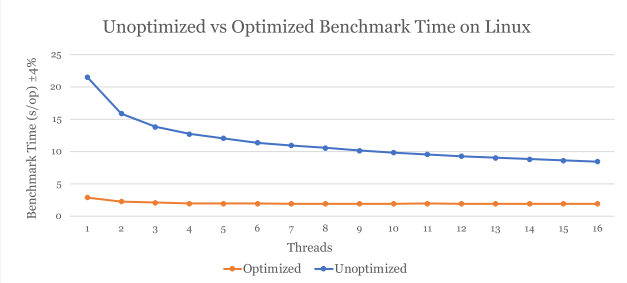

Figure 5 Benchmark Time on Linux for Unoptimized and Optimized

Figure 5 shows the difference in benchmark time before and after the *countNeighbour()* function is optimized, the latter runs 9.3x faster for 1 thread and 3.3x faster for 16 threads. The trend is alike for Windows and macOS.

There is almost no difference in time from the 4 threads onwards for the optimized version. Aside from the part that cannot be parallelised, this could be due to a communication bottleneck; as more threads are added, each worker spends more time communicating, which balances out the benefit brought by parallel programming.

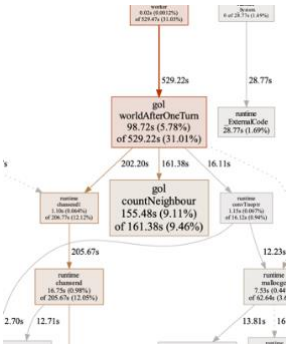

Figure 6 CPU Profile for Windows (Optimized)

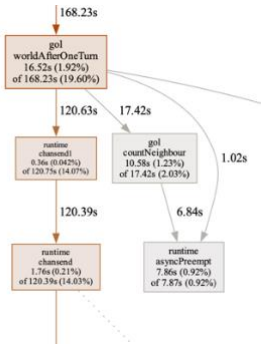

Figure 7 CPU Profile for macOS (Optimized)

The CPU profile for Windows (*Figure 6*) and macOS (*Figure 7*) also shows that the optimized version is much more efficient. The speed for computing the *worldAfterOneTurn* is 6.0x faster than before.

3. Critical Analysis and Potential Improvements

The implementation of keypress control for pausing and resuming (*keypress: "q"*) make use of busy waiting, where it repeatedly checks if another *keypress* for "*q*" to resume the game is received. This approach could waste the processor's power. It could be improved by using a mutex lock or a conditional variable. However, since the program pauses after "*q*" is pressed, using busy looping would not significantly affect the performance of this program.

4. Conclusion

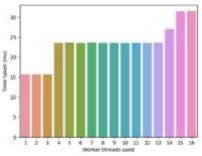

Figure 8 Linux Optimized 16x16

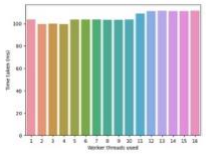

Figure 9 Linux Optimized 64x64

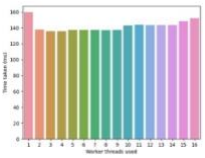

Figure 10 Linux Optimized 128x128

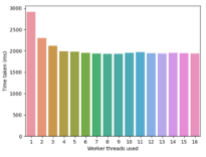

Figure 11 Linux Optimized 512x512

Figure 8 to *10* shows the benchmark time for different image size on Linux using the optimized version. Overall, there is a decrease in time from 1 worker to 2 workers, for larger images, this decrease is more rapid, and the graph shows a general decreasing trend; but for smaller images, the run time increases as more threads were added in; This might be where the communication time overtakes the benefit from doing the work in parallel.

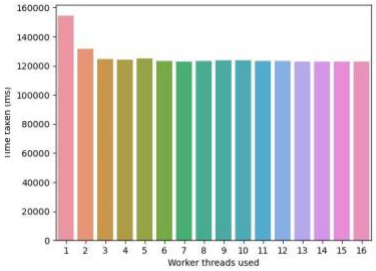

Figure 12 Linux Optimized 5120x5120

Figure 12 shows the benchmark time for running 100 turns for the 5120x5120 initial configuration. As a conclude, the program shows high potential for computing larger images with more threads.

Stage 2 - Distributed Implementation

1. Functionality and Design

1.1. Implemented Functionalities

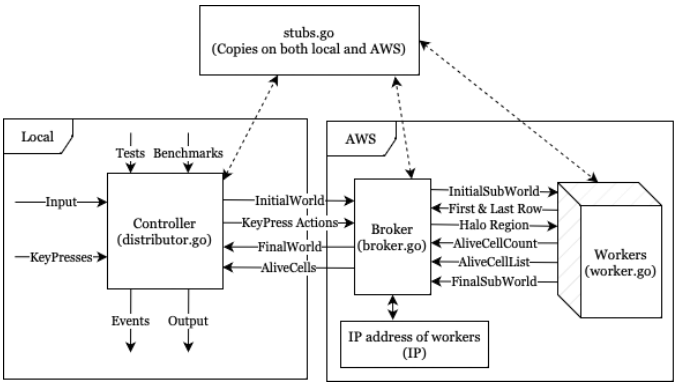

Figure 13 Distributed System Design

Figure 13 shows the design of the system of the distributed program (*The support of SDL display was written separately and will be discussed in 3.3*). The program was designed based on the publish-subscribe model that uses a broker as an intermediate to communicate between the local controller and workers on AWS nodes.

WorldStates (in Broker)	
CurrentTurn:	int
CurrentWorld:	[][]uint8
TotalTurns:	int
MutexLock:	sync.Mutex
PauseChan:	chan bool
Threads:	int
IpList:	[]string
WorkerList:	[]WorkerRecord
DistributedWorkDoneChan:	chan bool
WorkRequestSentChan:	chan bool
AllowChangeChan:	chan bool

WorkerState (in Worker)	
WorldSlice:	[][]uint8
Height:	int
Width:	int
RowAbove:	[]uint8
RowBelow:	[]uint8
MutexLock:	sync.Mutex
CurrentTurn:	int
StartY:	int

WorkerRecord (in Broker)	
WorkerNumber:	int
Connection:	*rpc.Client
WorldSlice:	[][]uint8
StartY:	int
EndY:	int
TopRow:	[]uint8
BotRow:	[]uint8

Figure 14 Data Structure Used

Figure 14 shows the data structure used for the broker and worker. The local controller sends the initial world and requirements to the broker at the beginning of the program; it was stored in the global variable, *WorldStates*, in the broker. Broker keeps track of individual workers using *WorkerList*, which composed of *WorkerRecords*. The broker will first invoke *workerInitialisation()* to set up all the workers by sending the initial sub-world to them. Workers will store the sub-world in the global variable, the *WokerState*, together with other setups. After that, the broker will iterate through all turns and request workers to compute the next state. Workers will only exchange the halo region with the broker to minimise the communication time. In addition, the local controller will send a request for *AliveCellsCount* that let the broker request *AliveCellCount* from every worker and respond to the local controller. The *keyPress* acts similarly.

1.2. Challenges Overcame

In every turn, *distributeWork()* is called as a goroutine for every thread to request worker compute the next turn. However, it could cause synchronous issues and race conditions with *AliveCellCount*, where workers are working on different turns. This was fixed by introducing *MutexLock* stored globally in *WorldStates*. It was locked before invoking the *distributeWork()* function to call workers and unlocked after all workers have responded and the *WorldState* has been updated. This makes sure that workers will respond correctly when *AliveCellCount* request is sent. *MutexLock* was also used when changing other fields of *WorldStates* to prevent race condition between different goroutines.

1.3. Potential Scalability

The broker connects to workers simply by reading in IP addresses in the *IP* file. Adding in new distributed workers is simple and convenient. It can be done by adding more IP addresses and ports in the *IP* file, restarting the program and specifying the number of threads accordingly, broker will be able to assign works to them.

2. Performance and Benchmarking

2.1. Test Environment

All benchmarking tests were taken under the Kernel Linux environment with Go version 1.19.1;

broker and workers are hosted on AWS nodes with Go version 1.18.6. To ensure stable connections, the computer was connected to the internet using an ethernet cable.

2.2. *Benchmark Time on AWS Analysis*

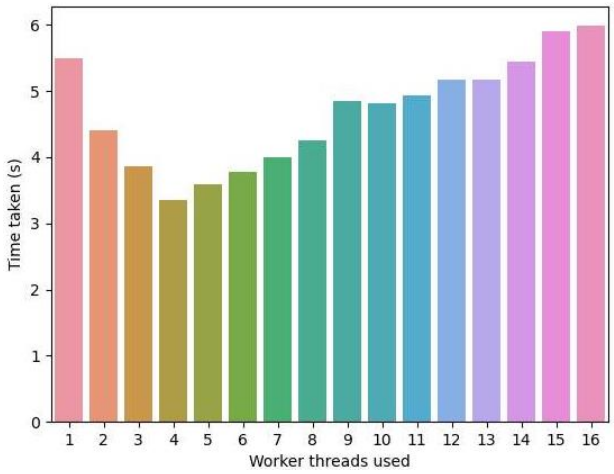

Figure 15 Benchmark Time for 512x512

Figure 15 shows the benchmark time of computing a 512x512 initial configuration for 1000 turns from 1 AWS node to 16 AWS nodes with the broker running on the AWS node. There is a rapid decrease of time at the beginning and a steady increase as the number of nodes increases from 4 nodes onwards.

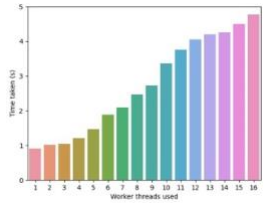

Figure 16 Benchmark Time for 64x64

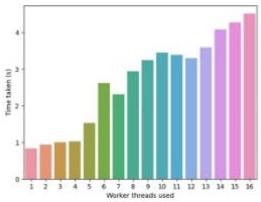

Figure 17 Benchmark Time for 128x128

By running for smaller images, the cost of communication becomes more significant. *Figure 16* and *17* show the benchmark time for 64x64 and 128x128 initial configuration with the same setup as 512x512. It shows a gradual increase in time as the number of nodes increases. This might be due to the cost of communicating between workers and brokers becoming more expensive when more workers were added in, and it also depends on the internet speed for sending and receiving data.

2.3. *Comparing the Benchmark Time Between Running on AWS and Local Ports.*

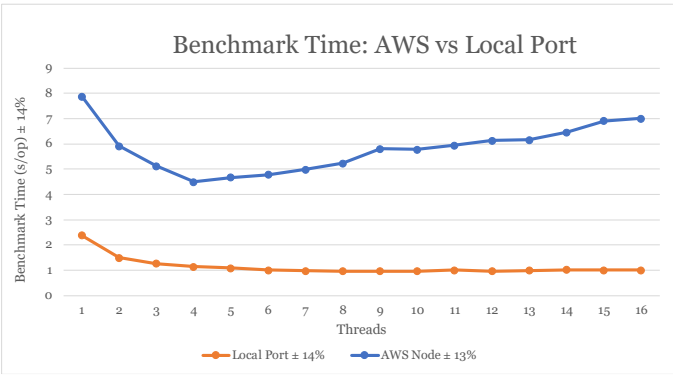

Figure 18 AWS vs Local Port Benchmark Time on Linux for 512x512

To understand how the internet connection could affect the performance of the program, *Figure 18* shows the benchmark time for running workers on local ports and AWS nodes. The benchmark time on local ports is how the program is expected to be. The increasing trend for running on AWS might be due to communication overhead, this will be discussed in part *3.1*.

2.4. *Go Trace Analysis*

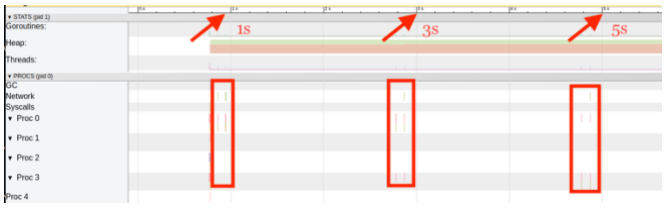

Figure 19 Go Trace for 512x512 on AWS

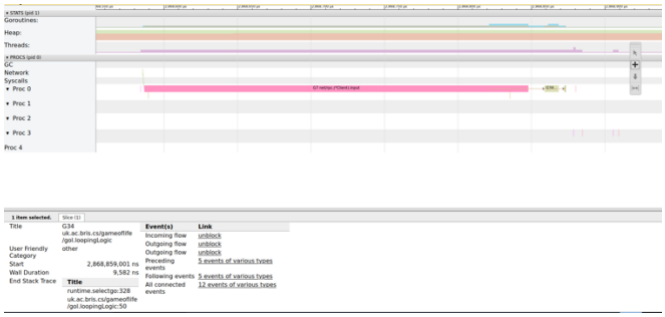

Figure 20 Zoomed Trace
(Zoomed in the 2nd red rectangle in Figure 19)

Figures 19 and *20* show the *Go Trace* for computing a 512x512 image remotely with the broker on an AWS node. By examining the trace, one could see that the only active goroutine during the program is the *AliveCellCount* which happens every 2 seconds; all other works were done remotely on the broker and workers over AWS,

and there is no unnecessary communication between the broker and the controller.

2.5. *Influence of Broker Connection*

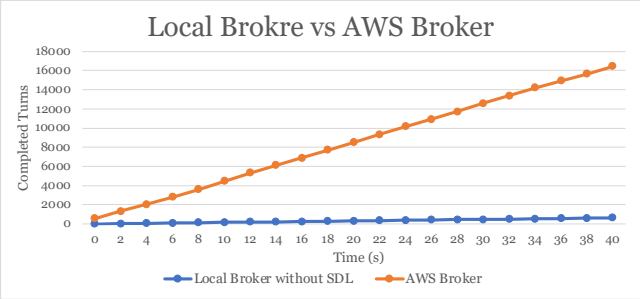

Figure 21 Runtime of Local Broker and AWS Broker for 512x512

Figure 21 shows a significant improvement in runtime when the broker is on AWS. This might be due to the connection between AWS nodes being much faster than connecting to local computers; it would complete 26x more turns than the local broker does. This not only reveals the importance of network connection on the performance of the program but also shows the importance of broker in this program.

3. Extensions and Potential Improvements

3.1. *Halo Exchange*

The implementation of the Halo exchange is like what we've done for the parallel part, where the halo region was sent separately with its sub-world. Thus, halo exchange was implemented, with the broker being the central distributor, from the start of the distributed part. Instead of sending the entire sub-world repeatedly, halo exchange promotes more efficient communication. However, there is still a communication overhead on the broker when updating the Halo region to workers. This could be improved by letting workers communicate their Halo region directly through *rpc* calls.

3.2. *Fault Tolerance*

3.2.1. Continuing the Game

Fault tolerance is implemented for the local controller to allow another controller to take over and continue the current game. When '*q*' is pressed, the current world state will be saved in the broker as *backUpWorldState*. When a new local controller starts, it can continue the previous world with command line flag: *-continue="true"* to continue the previous game or without the flag to restart. This allows switching the controller when faults occurred locally.

3.2.2. Auto Adaption on Number of IPs

We've implemented logic for the broker to adapt to the number of IPs provided in the IP file. When the user specifies more threads than the number of addresses reads, the broker will print a message to the console and continue the game with the number of addresses provided.

3.3. *SDL Live View*

We've turned the local controller both as a client and a server with *SDLDisplay()* registered. This allows the broker to call the local controller after each turn and sends the *FlippedCell* list as a request for it to be displayed.

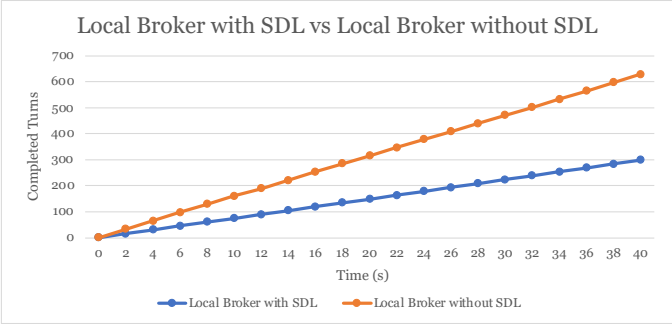

Figure 22 Runtime of With SDL and Without SDL on Local Broker for 512x512

Figure 22 shows that the program without SDL runs 2x faster because the broker now needs to communicate with the controller after each turn instead of every 2 seconds. This would worsen the communication overhead for the broker.

4. Conclusion

From the distributed part of the assignment, we've seen the benefit and the trade-off brought by distributing a serial program. It brings insights into the possibility of computing larger program with more distributed workers that is beyond the hardware limitation of a single computer.