Kaijian Liu tx22513
Siqi Yao aq22036

Introduction

This report presents an in-depth exploration of a parallel and distributed implementation of Conway's Game of Life in Golang that is a language renowned for its adept handling of concurrent processes. The core aim of this analysis is to provide an extensive and detailed evaluation of two implementations and the performance of the code. Furthermore, this report will identify and examine potential improvement, focusing on strategies for optimizing the code's performance and efficiency.

Stage 1 – Parallel Implementation

1. Functionality and Design

Firstly, this is a parallel version of the Game of Life, utilizing multiple workers for enhanced performance. Initially, we crafted a serial code that operates on a single thread when thread is one. In this mode, the entire game cycle is processed within a single thread, bypassing the need for multiple worker co-programs.

Moving to the parallel implementation, our focus was on concurrently computing different segments of the game's world. This is achieved by deploying multiple threads, each responsible for a portion of the computation and then aggregate these partial results, stitching them together into a cohesive outcome. To monitor the game's progress, we instituted a timer set at two-second intervals. Every two second, it safely reports the count of living cells.

To visually represent the game's state, we designed a function to output PGM images upon the completion of all turns. This involves commanding an I/O coprocessor for file writing, iterating over the game board for image data generation, and signaling the event channel upon task completion.

Lastly, for real-time interaction, we integrated SDL for game state visualization and a function to respond to specific key presses, such as 's' for outputting PGM image for current turn, 'q' for outputting PGM image for current turn and exiting the game, and 'p' for pausing and resuming. This responsive design enhances user engagement, offering direct control over the game's operational flow.

2. Benchmark and Critical Analysis

We conducted our benchmark tests using a university lab machine equipped with 6 cores and 12 threads. All data were collected on the same day, using this identical machine configuration for each scenario. This approach guarantees that our results are consistent and not influenced by variations in hardware performance.

2.1 Analysis Threads

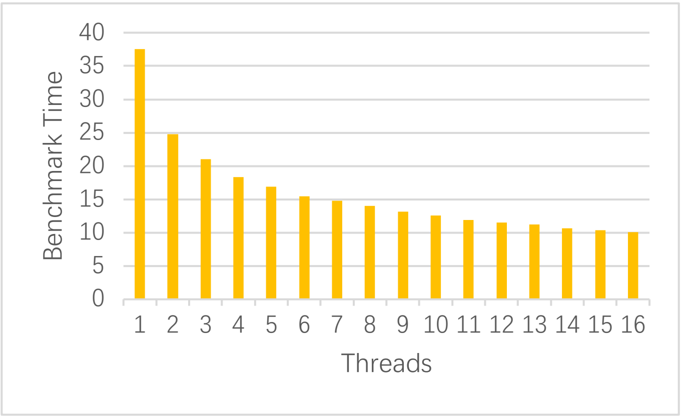

Figure 1: 512*512*1000 benchmark for different threads

According to Figure1, in this case we have changed the number of threads, from which different performance changes can be observed. We found that the processing is least efficient when it is single threaded. This is because all the computation is executed on a single thread. This leads to longer execution times, especially for processing larger images.

In a small number of threads (2-5) there is a significant performance improvement. This is because work can be executed in parallel on multiple processor cores. In a medium number of threads (6-12), performance will continue to improve, but not as dramatically as before. This is because the communication and synchronization overhead between threads increases as the number of threads increases. Diminishing marginal gains can be observed in the last large number of threads (13-16). This is because at this point the number of threads exceeds the number of physical cores of the CPU.

2.2 Analysis Different Images

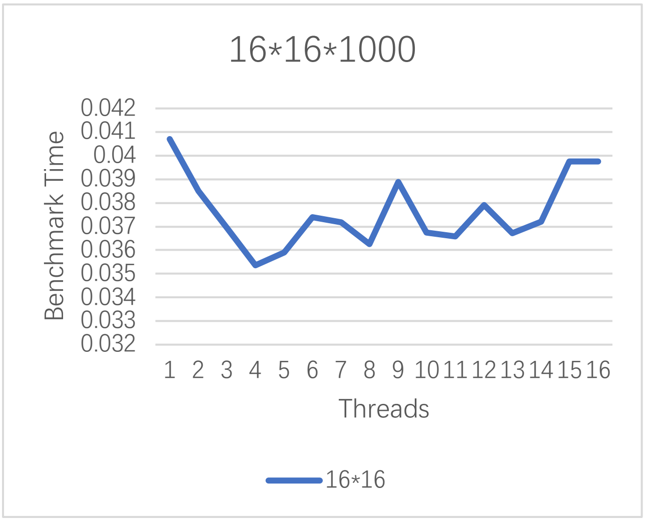

Figure 2: 16*16*1000 benchmark for different threads

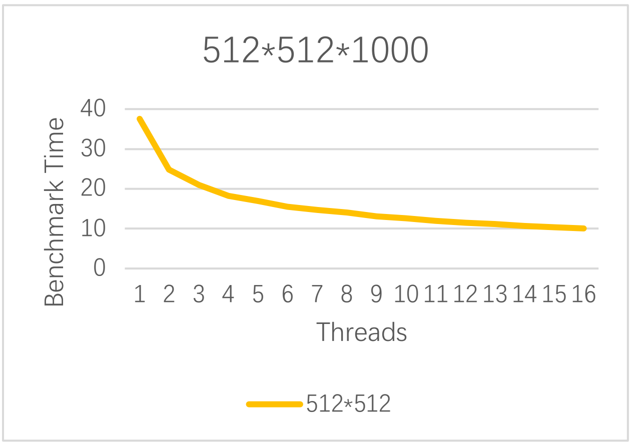

Figure 3: 512*512*1000 benchmark for different threads

When considering a constant number of threads across varied sizes of a gaming world, the impact on processing efficiency differs by size.

For smaller game worlds, (16*16), the additional

threads do not significantly enhance processing efficiency. This is attributed to the minimal computational demand of the task, where the overhead from managing multiple threads overshadows the benefits, given the task's low computational intensity. Conversely, in larger game worlds (512*512), the increase in data volume is substantial, and in this situation the advantages of parallel processing become most apparent. Benchmarking times significantly decrease as more threads are employed, signaling that multi-threading is indeed effective for reducing processing time and enhancing efficiency for expansive images.

Examining images of disparate sizes reveals a stark contrast in processing time variability. The smaller image size exhibits a variance of 2.16455E-06, indicating minimal fluctuation in processing time. On the other hand, the larger image size has a variance of 47.093, suggesting much greater inconsistency in processing duration. For diminutive images, multi-threading offers negligible performance enhancements due to the low computational load of the task and the predominant management overhead. In the case of larger images, while multi-threading can significantly expedite processing, the degree of acceleration tapers off as the thread count climbs, likely due to the inherent overheads and resource contention in a multi-threaded environment.

3. Optimization

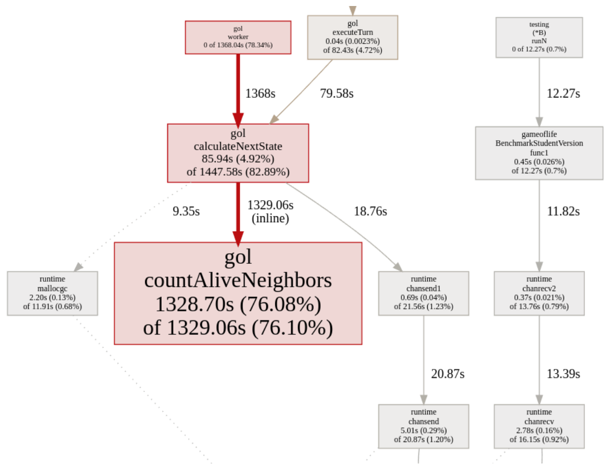
gol worker 0 of 1368.04s (78.34%)
gol executeTurn 0.04s (0.0023%) of 82.43s (4.72%)
testing (*B) runN 0 of 12.27s (0.7%)
1368s
79.58s
12.27s
gol calculateNextState 85.94s (4.92%) of 1447.58s (82.89%)
gameoflife BenchmarkStudentVersion func1 0.45s (0.026%) of 12.27s (0.7%)
9.35s
1329.06s (inline)
18.76s
11.82s
runtime mallocgc 2.20s (0.13%) of 11.91s (0.68%)
gol countAliveNeighbors 1328.70s (76.08%) of 1329.06s (76.10%)
runtime chansend1 0.69s (0.04%) of 21.56s (1.23%)
runtime chanrecv2 0.37s (0.021%) of 13.76s (0.79%)
20.87s
13.39s
runtime chansend 5.01s (0.29%) of 20.87s (1.20%)
runtime chanrecv 2.78s (0.16%) of 16.15s (0.92%)

Figure 4: Benchmark before performance optimization

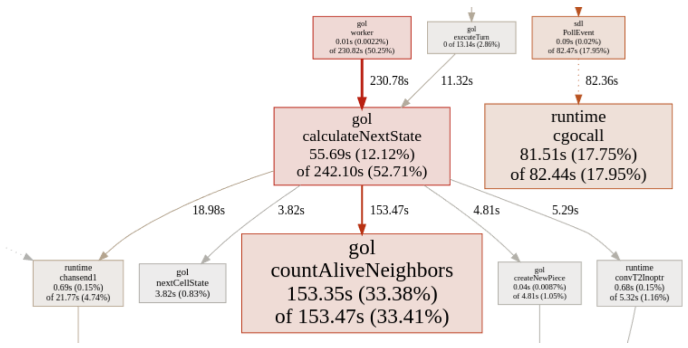
gol worker 0.01s (0.0022%) of 230.82s (50.25%)
gol executeTurn 0 of 13.14s (2.86%)
sdl PollEvent 0.09s (0.02%) of 82.47s (17.95%)
230.78s
11.32s
82.36s
gol calculateNextState 55.69s (12.12%) of 242.10s (52.71%)
runtime cgocall 81.51s (17.75%) of 82.44s (17.95%)
18.98s
3.82s
153.47s
4.81s
5.29s
runtime chansend1 0.69s (0.15%) of 21.77s (4.74%)
gol nextCellState 3.82s (0.83%)
gol countAliveNeighbors 153.35s (33.38%) of 153.47s (33.41%)
gol createNewPiece 0.04s (0.0087%) of 4.81s (1.05%)
runtime convT2Inoptr 0.68s (0.15%) of 5.32s (1.16%)

Figure 5: Benchmark after performance optimization

According to the Pprof analysis (Figure 4), it is found that the CPU time for the countAliveNeighbors function is 1328.70 seconds, which occupies 76.08% of the CPU time. In the current implementation, boundary checking is performed for each neighbor of each cell and modulo operations are used to handle boundary cases. Since modulo operations are relatively expensive operations, there are some faster alternatives such as per-bit operations can be considered, especially when the divisor of the modulo operation is a power of 2, since the height and width of the image being processed are powers of 2 (e.g., 16, 64, and 512). Bitwise operations are typically faster than modal operations because they are processor-based bitwise operations that do not perform division. After replacing modulo arithmetic with bitwise arithmetic, the CPU time for the countAliveNeighbors function drops to 153.35 seconds (Figure 5). This means that the improvement in performance is quite significant, with a performance increase of about 8.66 times.

In the calculateNextState function, the performance improvement is also noticeable, but not as prominent as the countAliveNeighbors function, which improves by a factor of about 5.98.

4. Potential Improvements

For the caluculateNextState function, it consumes a total of 52.71% of CPU time due to its direct and indirect calls. Optimizing this function can lead to significant performance gains. Firstly, it can be considered whether all cells need to be calculated, for example if a cell and its neighbors did not change in the previous turn then this cell will not change in this turn either, so the state calculation for these cells can be skipped. In addition, if the state of cells in a region is detected to be completely stable then the computation for that region can be skipped.

5. Conclusion

To summaries, this parallel implementation shows excellent performance benefits for the Conway Life game. By testing the performance in both single and multi-threaded scenarios, the experimental data clearly demonstrates that the performance improvement decreases as the number of threads increases, but it shows adaptability in different scenarios.

Stag 2 – Distributed Implementation

1. Functionality and Design

At this stage, our goal is to create a broker that uses multiple AWS nodes to collaboratively calculate the new state of the Game of Life board and pass that state between machines over the RPC work.

An initial world is first entered into distributed and information about the initial world is then passed to the broker. The broker is then split into multiple servers to process this information separately. The server then passes the processed information back to the broker, where it is merged. Finally, distribute the complete copy to Distributed. That's how the program works.

We first made sure that the single-threaded and single-machine implementations were feasible and split them into two components. We then implement a basic controller that allows the logic engine to evolve a specified number of turns of the Game of Life through blocking RPC calls. This lays the foundation for subsequent distribution.

We then implemented a reporting mechanism to report the number of alive cells to the local controller every 2 seconds and implemented communication between the local controller and the Gol engine through RPC calls. Then we focus on enhancing the capabilities of the local controller. We enable it to output the final state of the game board of life in the form of a PGM image.

We also implemented the management of the Gol engine through the controller. It introduces rules that include generating PGM files when you press the 's' key, gracefully shutting down the controller client when you press the 'q' key, shutting down all components when you press the 'k' key, and pausing or resuming processing on the AWS node when you press the 'p' key. This allows users to control and interact with distributed systems.

After implementing the keypress, we need to split the Game of Life board state calculations across multiple servers and collect the results in one place. We do this by creating a program called broker. It receives information about the initial world from the controller, and then we distribute the task to multiple servers by using RPCs instead of channel calls to collect and aggregate server responses for distributed collaboration.

2. Testing and Critical Analysis

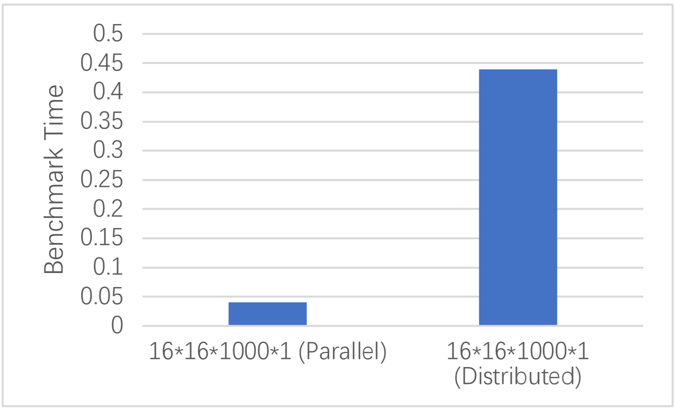

Figure 6: 16*16*1000 benchmark for different version

Two different execution strategies, parallel and distributed, are compared separately based on the test data. 1000 iterations are executed for each of the two different sized datasets (16*16 and 512*512).

Firstly, according to Figure6, it can be concluded that the dataset size has a significant effect on the runtime, for smaller datasets the parallel version is much faster than the distributed version (about 10 times faster). This is due to the fact that the communication and synchronization overhead for small datasets is not as great as the data processing itself. Therefore, the efficiency from parallelization is more prominent. However, there can be some fixed overheads in distributed policies, such as task allocation, result collection, or RPC calls. In small datasets, this overhead is relatively high.

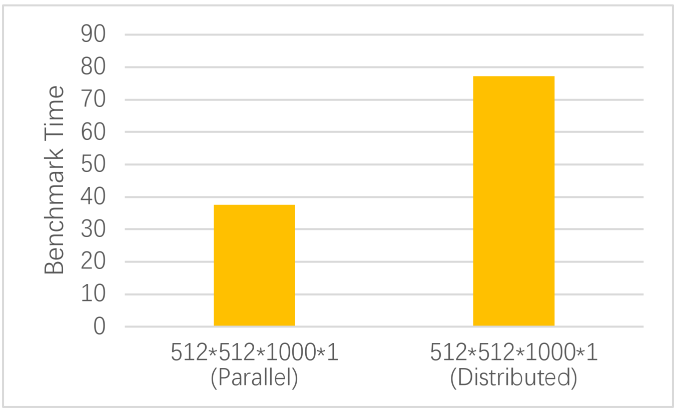

Figure 7: 512*512*1000 benchmark for different version

However according to Figure7, when the dataset size comes to 512*512, parallel grows significantly faster than distributed. Although distributed did not outperform parallel in this benchmark test, distributed will have a greater performance advantage as the dataset grows larger.

3. Potential Improvements

Based on benchmark comparisons, parallelization is the best option when dealing with small datasets. Because it avoids the extra overheads associated with distributed computing. But in large-scale datasets, while staying ahead, it also shows that the distributed strategy hints at its potential improvement on distributed systems. To avoid the overheads associated with communication and to improve the efficiency between individual nodes, we can implement halo exchange. Where each turn only needs to communicate edge rows between nodes each and does not need to be with the central distributor node after each iteration. this means that sending a large amount of data per turn can be avoided, and for large scale datasets it can be faster for large-scale datasets and improve the performance. In implementing halo exchange, the message passing logic needs to be designed to ensure that the data is synchronized as well as correct, and special attention needs to be paid to the boundary conditions.

Another possible improvement is to implement a parallel distributed system. This means that each machine has multiple threads, and computational tasks can be split into smaller chunks that are executed simultaneously. Parallel distributed systems have good scalability because of the ability to maintain a linear increase in performance as workers are added. Maximizes the use of server-side computing resources such as number of CPU cores, memory. Ideally, the performance growth of the system should be proportional to the number of computational resources added.

4. Extension

SDL visualization has been successfully implemented in the distributed version. It can show data and state changes in distributed implementations, but the performance degradation is noticeable due to the need to pass CellFlipped in each turn, which contains a huge communication overhead that makes the displayed SDL image delayed. One of the problems encountered in solving the SDL visualization extensions was communicating over a closed channel thus in order to be able to pass all the tests.

5. Conclusion

In summary, the comparison between parallel and distributed execution strategies indicates that the impact of dataset size is crucial. As distributed execution increases with the size of the data set, its potential performance benefits become more apparent. Ultimately, the choice between parallel and distributed strategies depends on the specific characteristics and scale of the datasets under consideration.