Emil Centiu: zl18810
Valentin Popescu: bk18529

Report

Coursework 1 Weeks 5-10

Concurrent Computing COMS20001

Introduction

The aim of this report is to showcase our "Game of Life" implementation. It will contain two parts. The first one implies functionality and design and the second one tackles the analysis of our experiments. We will start with presenting the challenges we faced during development process and the overall evolution of the project.

Functionality and Design

Stage 1a

The first step was to implement the "Game of Life" logic. Determining what cells must be changed was done by iterating them through the matrix. Firstly, the coordinates of the eight neighbours of each cell were determined using two directional variation arrays (`dx`, `dy`). For example: the top left neighbour's coordinates are accessed by choosing `dx=-1` and `dy=-1` and adding them to the current cell coordinates. This method provided access to all neighbours of a cell and possibility to check their state (alive/dead). As the number of alive neighbours for each cell is kept in a variable. It is simple to check if there are changes to be done. Thus, we added all the cells that must be changed in an empty slice. Then we went through the slice and flipped dead cells into alive ones and vice versa.

Finally, we introduced the communication between the `distributor` and `writepgmImage` to output the changed matrices.

Stage 1b

We needed to parallelize the sequential version of stage 1a by creating a new set of functions. The main challenge was to divide the matrix correctly for each worker to take their part. The `worker` function contains the game logic. It receives the initial configuration for his part of the matrix though a channel from `sendData`. This function is responsible for sending the coordinates of the alive cells to each worker as well as assuring the matrix circularity. The `update` function flips dead cells in alive and alive cells to dead. The `distributor` acts like a "main" function now.

Another setback encountered was that all workers were sending on the same channel and the updated matrix was not the one desired. This happened because some workers were finishing their parts faster than others. To solve this in the `distributor` was created a slice of channels, one for each worker. Now each worker sends on his channel. Both the worker and the update function are called as go routines. The `distributor` synchronizes the workers, waiting after each turn for them to finish their job. This was implemented using a `WaitGroup`.

Stage 2a

Stage 2a was implemented using a select statement in the `distributor`. Depending on which key (p,q or s) is sent on the key channel, the program behaves differently. To implement the "quit" and "pause" commands, two Boolean variables were used. For the "screenshot" command a new function (`printPGM`) was defined. It sets the I/O command to output, gives the file name and sends the current configuration of the world to the `pgmio` goroutine.

Stage 2b

To solve this stage, a counter variable was used in `distributor` to save the initial number of alive cells. In the `update` function a variable was used to keep track of the number of changes. A

transformation from dead to alive was increasing the variable value while a transformation from alive to dead was decreasing the value. This value was sent through a channel to the distributor which was updating the total number of alive cells after each turn. To output the result every two seconds a ticker object was initialized. In a select statement a print command was done when receiving on the ticker channel.

Stage 3

The challenge when having a number of threads/workers (6,10,12) that is not a divisor of the image height is that it remains a remainder. This was solved by adding the remainder to the last worker.

Stage 4

The purpose of this stage is to increase the responsibility of the workers by modifying them to do the update as well. For this, we needed a halo exchange scheme where each worker had a channel in which it receives the halos from above and below (form neighbour workers). Then, it needs to wait after the other workers to finish the turn before sending the halos to other workers. We implemented that using semaphores. Also, the I/O commands had to be executed at the end of each turn.

Stage 5

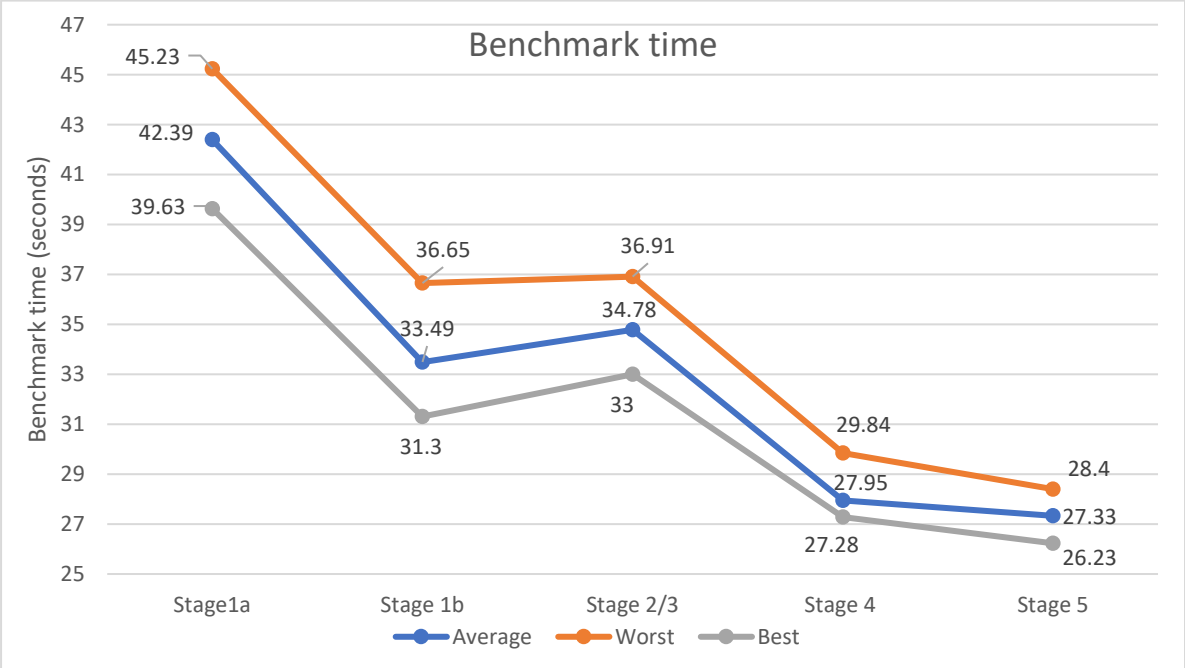
The main difference from stage 4 is that now the workers have access to the whole matrix. Therefore, they only need to wait for each other before updating, so that the halos that are used by the other workers are not broken.

Tests, Experiments and Critical Analysis

Because performance was a very important aspect since the beginning of the project, we have implemented an efficient version of “Game of Life”.

Benchmark analysis

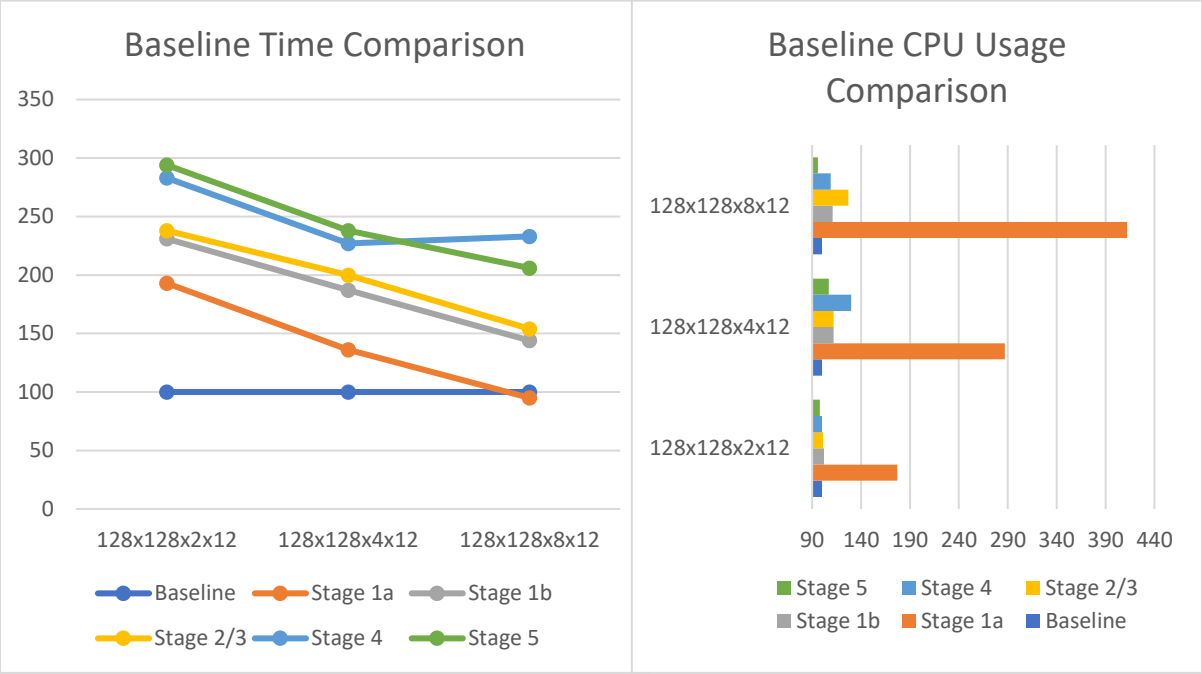
For each stage we ran 10 benchmark tests in order to determine a more accurate execution time. The results are in the graph below:



Important differences can be seen between stage 1a and 1b and between stage 3 and stage 4. Those parts are exactly the parts where we implemented a performance improvement. The performance of stage 1b, 2 and 3 is similar because we didn't implement any concurrent features, we just added more functionality. Stage 1b is the concurrent version of stage 1a while stage 4 uses the workers even more, those being responsible for updates as well now. Stage 5 shares the same implementation but now the workers have access to the whole matrix, and therefore no communication between them is needed apart from the semaphores after each turn. We can also see that the average time is closer to the best time for all stages, meaning that the worst time is more of an exception.

For Stage 1a, we tried to minimize the bottleneck form the communication by only sending the alive cells' coordinates instead of the whole matrix and only the coordinates of cells that are changing their value back to the distributor. We pushed the same idea into stage 4, and because there aren't that many alive cells in the halos, we didn't expect much performance difference and it turned out to be true.

Baseline comparison



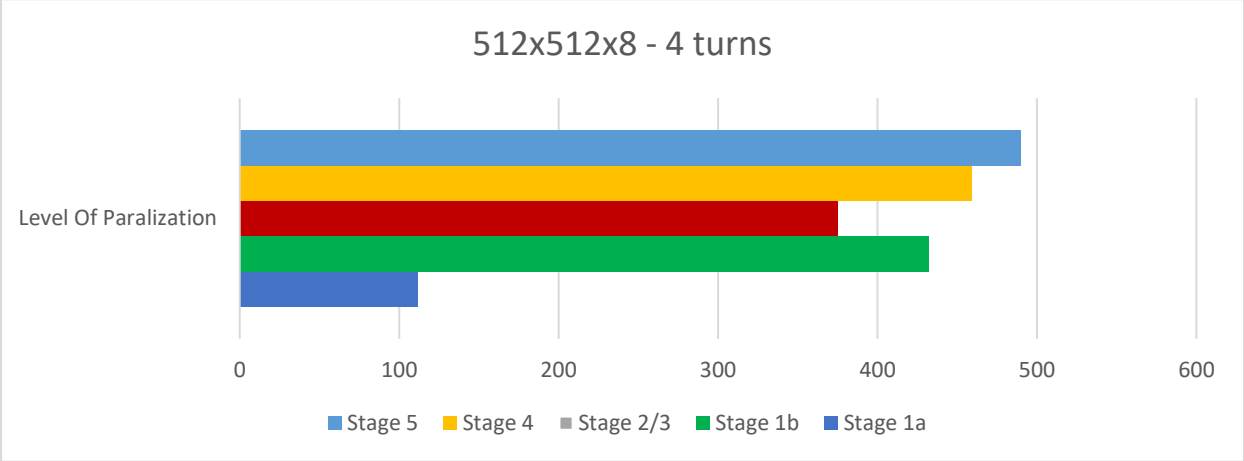
CPU profiling

Stage	Most time-consuming Function	% CPU Usage
1a	Distributor	83
1b	Worker	76.08
2/3	Worker	71.96
4	Worker	88.34
5	Worker	90.67

As more and more work is moving for the distributor (Stage 1a) to the worker, worker is getting more and more time, apart from the stages 2 and 3, where there was no change in algorithm.

Time

This command was particularly useful because it enabled us to measure the level of parallelism of our stages compared to the baseline.



The graph reflects what we would expect, stage 1a being the least concurrent one as it is doing the logic sequentially. Stage 1b is a big leap in terms of parallelism, but it is less than stage 4 as it bottlenecks at the communication between distributor and workers. Stage 4 is more parallel because there is less waiting in communication, and stage 5 is the best, as we are sending the reference to the matrix generated in distributor.

Conclusion

During this project, we found that a more parallel program is not necessarily better. There can be bottlenecks in the communication between goroutines and there is the possibility that a concurrent program can run even worse than its sequential version. However, if those are tackled properly, there are significant performance improvements (in our case, up to 2x form stage 1a to 5).