if20376

uo20643

Stage 1 – Parallel implementation

1.Functionality and design

1.1. Task Overview

The core idea of the parallel assignment is to evolve the Game of Life board for the desired number of turns, and then save the last iteration of the board as a PGM file. The implementation should allow the user to interact with the board by either halting the execution, pausing the execution or saving the current state of the board as an image.

1.2. Challenges

The trivial solution to the task would be to read the image that contains the board, then sequentially compute the next state until we reach the last turn, when we save the board as a PGM file. However, this is suboptimal, due to the following reasons:

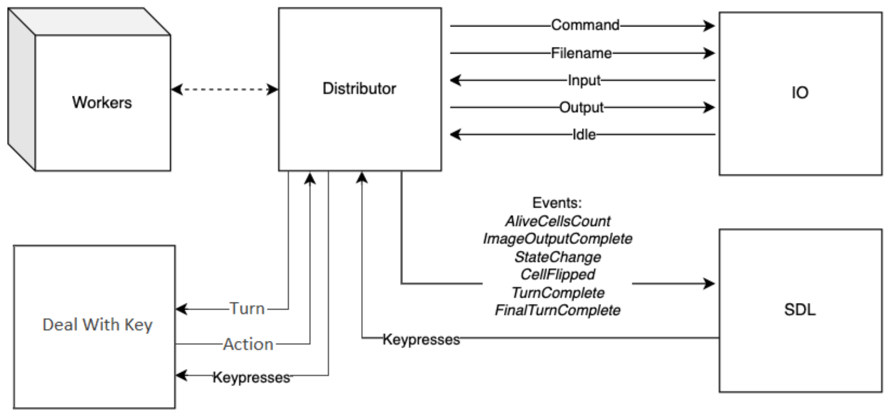
- The solution works well only when dealing with rather small images for a small number of turns. The implementation without a doubt becomes impractical when thinking of a 5120x5120 board that should be evolved for 1000 turns.
- The user wouldn't be able to interact with the board. In sequential programming there is no such thing as pausing the execution of a program until told otherwise by the user. This means that the only interaction the user has with the game is being able to see the output when the board is saved.

1.3. Design choice

Given the constraints that must be met, the entire program needs to run in a parallel fashion, hence the need to use concurrent programming. We will make use of the processor threads to minimize the time spent on evolving a single turn for the board. This design also solves the issue of user interaction, since we can now deal with both key presses and the actual playing of the game.

Since we may need to exchange information between threads, another decision had to be made: should we use the shared memory model, or the message passing one? We thought it would be best to use the message passing model, since channels already use some form of memory sharing and locks. This means we can avoid having global variables, therefore only allowing certain functions to access certain variables.

Apart from the goroutines that already existed in the diagram provided in the skeleton, we decided to add another one: dealWithKey. The distributor forwards the keypresses channel to this function. dealWithKey runs on an infinite loop and checks whether there's any keypress. If there is, it lets the distributor know of it using a channel, sending which action type was requested: pause, save, or quit. The distributor checks at the end of each iteration if dealWithKey sent any action down the channel. Some types of actions need the program to print the current turn and this is passed from the distributor to dealWithKey using another channel.



1.4. Implementation

Our implementation of the parallel version of the program tackles all of the steps presented and implements the ability of doing keypresses while paused, too. All the Game of Life logic is found inside the distributor function. It begins by sending the ioInput down the ioCommand channel, which triggers the io goroutine to open the specified file and start reading the image byte by byte. Based on the input parameters, it creates a filename and sends it down the ioFilename channel.

Once the image has been read, the algorithm is being applied for the number of turns specified by the parameters. The board is broken into height/threads slices which are then sent to the algorithm function, however as goroutines, thus each slice is transformed in a parallel manner. The output of each of the workers is then sent down its respective channel, thus preventing the mix-up of data. The order in which we piece back the board is given by the order in which each worker got its part of the board. Although there may be some workers ready to deliver the result, when appending, we wait for the information to come in the same order it was sent. In the end, we update the old world with the one retrieved from piecing together the goroutines results, and we let the SDL know that a turn has been completed by sending a TurnComplete event down the events channel.

In each evolution of the board there is a new world that is created. This is done because each cell updates its state based on the neighbors' values, thus making the necessity of having the previous world stored while putting the updated states on a new one. Otherwise, by changing the value of a pixel in the current board, we will produce the wrong value for its neighboring pixels.

While the distributor function does all the things mentioned, there are other processes running in the meantime. The dealWithKey goroutine constantly checks the keyPresses channel and informs the distributor by using a channel whether it needs to take a screenshot, halt or pause the execution of the turns. Then, before updating the board to the new one, the distributor checks whether there is any action it needs to take. The way it does this is by using a select statement, which doesn’t block on receiving, therefore allowing the distributor to keep on working if there isn’t anything else it needs to do. The extra functionality we implemented here is that when pausing the execution, it is possible to either take a screenshot or quit. The way we’ve done this is by calling another function when the key P is pressed: another infinite for loop which takes the input from the keyPresses channel, however what is different is that when P is pressed again, it sends a value down a channel which lets the distributor continue doing the work (the first P lets the function know that it needs to way for a value from a specific channel before continuing). Before updating the world, we also use a select statement which verifies whether the ticker has ticked, in which case we send a AliveCellsCount event that lets the user know every 2 seconds the turn and number of alive cells. The reason we used a select statement here is that otherwise the world would have been updated every 2 seconds, which would make the program extremely slow.

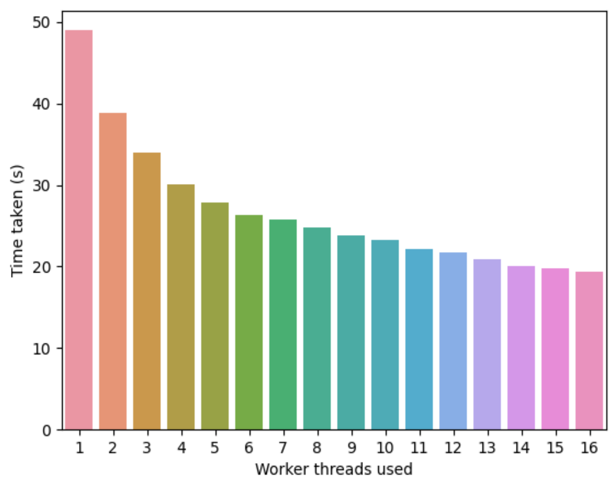
At the very end of the distributor, we let the SDL know that the final turn has been completed and we close the channel so that it can know that nothing else will be sent, so there is no need to wait for events.

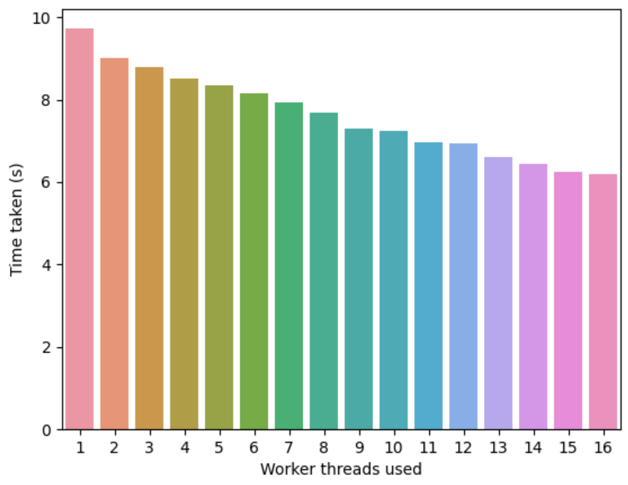
1.5. Optimizations

The first iteration of our implementation was a single threaded one, in which the next board state would be computed in a serial fashion. The next iteration had an upgrade, providing a parallel implementation. The distributor function sends the board to several goroutines, attributing them the range of Y coordinates they must do computations on. The goroutines then send back a slice containing the updated part of the board they've been tasked with.

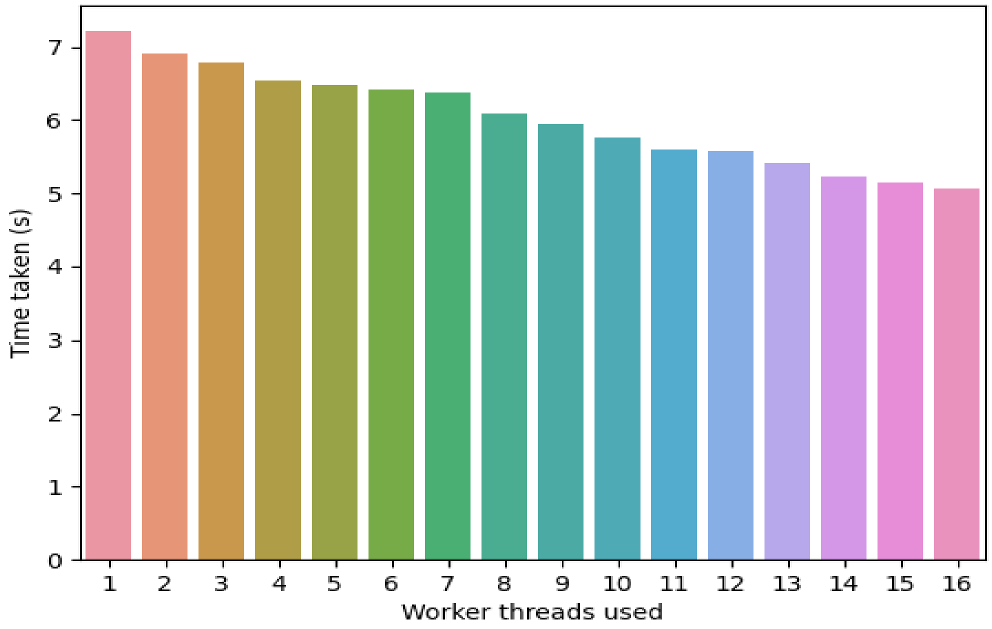
1.5.1. The algorithm uses the modulo operation to wrap around the 2d array and give the impression of an enclosed space. This operation is rather expensive, and an optimization we found for this has something to do with the fact that the dimensions of the board are powers of two. If the input's dimensions are such numbers, the modulo operation can be replaced by a bit shift operation, which we enclosed in a function called wrap. Below can be seen the how major the improvement is, and how the multithreaded implementation runs faster than the single threaded one.

Note that the benchmark plots provided below are generated by evolving a 512x512 world for 1000 turns on a Linux Virtual Machine using 4 of the cores of an Intel Core i7-10750H CPU

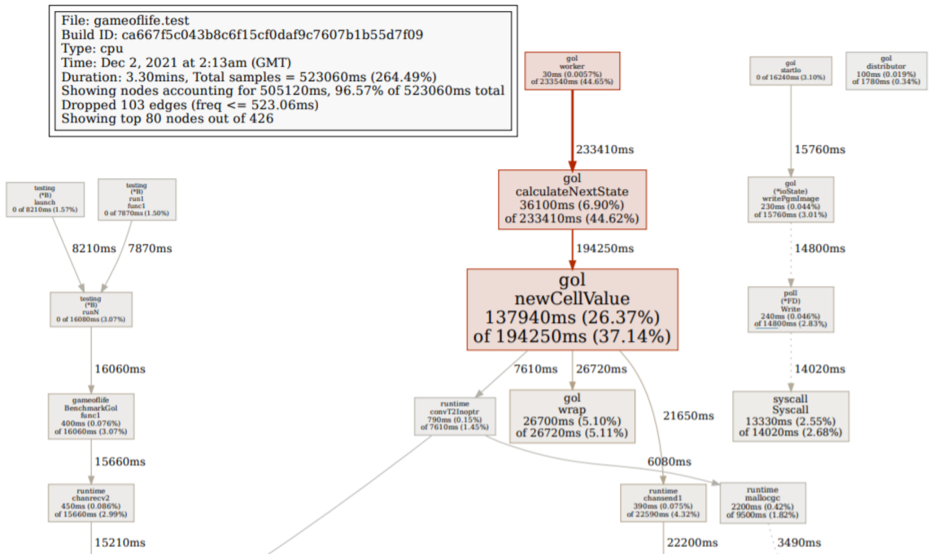




1.5.2. Another optimization that we've done has to do with the way communication was done between the distributor and the workers. As presented above, the distributor used sent the workers the whole board to allow them to properly find the lines above and below the bounds they were given. This has been changed and thus each worker only gets the amount of data it really needs: the part of the board it's working on and 2 extra lines, one above and one below, that will help it make the computation for the cells that are on the top and bottom rows. Below there can be seen how this optimization helps the runtime in comparison with the previous iterations. The runtime scales reasonably well with the number of threads but because the overhead caused by sending/retrieving data and dealing with IO is close to the computation needed for updating the board, the results aren’t exceptional. On bigger boards the improvement would be more obvious, but far from halving each time the number of threads is doubled. As stated above the machine on which the benchmark was done only had 4 available CPU cores, anything above that was done virtually using hyperthreading and go process scheduling.



In the below snapshot, coming from the CPU execution profiles, it can be seen that the newCellValue function, which iterates through an individual cell’s neighbors and returns whether the cell will be dead or alive, is the one that takes most of the time to run, which is an expected behavior since it runs Width * Height times for each evolution.



```
File: gameoflife.test
Build ID: ca667f5c043b8c6f15cf0daf9c7607b1b55d7f09
Type: cpu
Time: Dec 2, 2021 at 2:13am (GMT)
Duration: 3.30mins, Total samples = 523060ms (264.49%)
Showing nodes accounting for 505120ms, 96.57% of 523060ms total
Dropped 103 edges (freq <= 523.06ms)
Showing top 80 nodes out of 426
```

gol worker 30ms (0.0057%) of 233540ms (44.65%)
233410ms
gol calculateNextState 36100ms (6.90%) of 233410ms (44.62%)
194250ms
gol newCellValue 137940ms (26.37%) of 194250ms (37.14%)
7610ms
26720ms
21650ms
runtime convT2Inoptr 790ms (0.15%) of 7610ms (1.45%)
gol wrap 26700ms (5.10%) of 26720ms (5.11%)
6080ms
runtime chansend1 390ms (0.075%) of 22590ms (4.32%)
runtime mallocgc 2200ms (0.42%) of 9500ms (1.82%)
22200ms
3490ms
gol startIo 0 of 16240ms (3.10%)
gol distributor 100ms (0.019%) of 1780ms (0.34%)
15760ms
gol (*ioState) writePgmImage 230ms (0.044%) of 15760ms (3.01%)
14800ms
poll (*FD) Write 240ms (0.046%) of 14800ms (2.83%)
14020ms
syscall Syscall 13330ms (2.55%) of 14020ms (2.68%)
testing (*B) launch 0 of 8210ms (1.57%)
testing (*B) run1 func1 0 of 7870ms (1.50%)
8210ms
7870ms
testing (*B) runN 0 of 16080ms (3.07%)
16060ms
gameoflife BenchmarkGol func1 400ms (0.076%) of 16060ms (3.07%)
15660ms
runtime chanrecv2 450ms (0.086%) of 15660ms (2.99%)
15210ms

3. Reflections and Conclusion

After having completed this stage, several conclusions can be drawn. The most important one, however, is that this task has opened a new layer of responsibility. We as programmers have much more duty than just writing a sequence of functions which in the end output the desired result. It would be a waste not to use every resource to its potential, since this is what minimizes the total runtime. Nonetheless, this raises a lot of other issues, such as avoiding deadlocks, protecting critical code sections, and the hardware the software will be run on, as we can then make decisions which will benefit the most that architecture.

Stage 2 – Distributed implementation

1. Functionality and design

1.1. Task Overview

The core idea of the distributed assignment is to create a client-server communication over a network. The client reads the image and sends it to the server, which then applies the same Game of Life algorithm from the parallel assignment, and then send the final board state as some form of a reply.

1.2. Challenges

The biggest challenge is that there is no shared memory to which both the client and the server have access. This is an issue considering that the channel communication model we used to exchange information when need be doesn't work anymore. Another thing we had to consider was that the implementation of key presses will be much harder, as there is no way we can be certain that the server receives the command on time (i.e. if we wanted to take a screenshot of the board on the 20th turn, but we end up receiving a screenshot of the world in the 31st turn).

1.3. Design choice

Thankfully, golang's net/rpc package solves all the mentioned issues. By using remote procedure calls, we create a level of abstraction between the client and the server. It takes away the 'distributed' concept and treats the execution as though it happens on the same machine.

1.4. Implementation

As far as implementing the design goes, the core ideas from the parallel stage stay the same, apart from communication. We managed to make the Game of Life logic run on a single, multithreaded AWS node.

We begin by reading the board, and then we run 2 goroutines which interact with SDL, by reporting the alive cells number every 2 seconds and dealing with input keys. After that, we dial the server using the parameter server address. The world received as input then uses the makeCall method, which calls the server and uses the function provided in the stubs package to tell the server what to do.

In the meantime, the goroutines also call the server, one of them does this every 2 seconds to get the number of alive cells and the current turn so that it can be reported to SDL, and the other does it every time an input key is received so that we can get the current turn and the board when saving, pausing, closing the client or killing both the client and the server. The reason we need both the board and the turn is that when we either screenshot or quit we need to report the final number of alive cells, and the evolve method doesn't return that unless all the said turns are played.

The server listens on the port 8030 if no other port is given as a flag. It then registers the methods provided in the stubs package, and starts listening on the specified port. Once a client dials that address, the server starts accepting it and responds to any requests the client has.

Using global variables and channels, and a mutex lock, is what allows the server to report back the number of alive cells and to obey the commands given by the keys, and also helps with the synchronization of the multiple Remote Procedure Calls the server will get. The Evolve function is the one that does all the Game of Life related logic and when all turns are completed or a quit command is sent, sends the world back to the client as a response. The critical code section here is when updating the previous world to the current one and increasing the turn counter. Not using a mutex lock here would cause a race condition, as there is the possibility that the client makes a call to retrieve the number of turns and alive cells, but at the same time evolve is updating the values, thus getting erroneous results. All the other methods which retrieve said information are using the same lock, so that when one of them has to report back to the client, the evolution is briefly stopped to allow that.

Since the rpc.Accept() function is a blocking one, we had to use it as a goroutine in order to kill the server when need be. The use of a global channel tells the server when it can be closed. When the client receives the key K as input, it makes a call to the server which puts something down the 'kill' channel. The main method then retrieves that information, waits for a bit to allow the client to close first, and then uses the Close() function to stop listening on that port.

2. Reflections and Conclusion

The distributed stage of the coursework proved to be a more significant task than the parallel one. We managed to get all done up until stage 4, implementing the keypresses part. The most difficult part about the task was managing to synchronize the multiple Remote Procedure Calls the client send to the server: they all had to use the data one of the goroutines was working on, and thus we decided to use shared memory between the goroutines. This seemed easy at first, but made us realize how cautious we needed to be, especially when working in a distributive manner. We learned more about programming, how complex the issue of distributing work across a networks is and how error prone working over the network can be.