Programmeringsuppgift 2

Getting started

All submissions should be implemented in Go unless the problem specifies something different. You can download Go at https://go.dev/dl/. You can also use various package managers to install it, e.g., HomeBrew on macOS.

Cite any relevant sources or approaches that you used in order to understand and solve the problems.

Objective

A) Locking strategies. This part is about running experiments on your computer working with a Linked List in order to understand the effect of three various locking strategies.

B) Implement a sorting algorithm using concurrency in Go

Part A - Locking strategies

You will implement three different locking strategies. Make sure that you separate the code for each solution so they can be run/shown independently.

Waitgroup

Use a Waitgroup for your goroutines. Do not use channels in this exercise.

These functions *must* be implemented and working correctly

Add(), Remove(), Contains(), find()

Use Contains() to check if a node with a certain value was added or not.

Functions that you can implement that might help you

count() - function that counts the number of nodes in your list. Use it for checking how many nodes you expected to add to the list vs how many was actually added.

strictly_increasing() - function that checks if the next node is larger than the current node as you are traversing the list. Ensure no duplicate key-values.

Problem A.1 - Starting point

Linked List with a set

a) Implement a Linked List to hold nodes with unique key-values (a *set*) that is designed for sequential access. Use the structure discussed in the lecture.

b) Try to add a couple of thousand unique nodes to the list using your Add()-function. Compare adding nodes using only one (1) goroutine with two (or more) goroutines and run the experiment several times. What do you expect to see? What is the cause of the observation?

Briefly describe your implementation and your findings in the report.

Problem A.2 - Coarse- and fine-grained

Number of cores for the experiments: Run each experiment with an increasing number of goroutines, up to 32 times the number of cores in your computer. If you have 4 cores the number of goroutines should be increasing like this: 1, 2, 4, 8, 16, 32, 64, 128 goroutines.

A.2.1 Coarse-grained locking

a) Build upon solution in A.1 and implement a thread-safe *coarse-grained locking* Linked List solution, (using the same lecture material).

b) Add 10 000 nodes with *randomly* numbered values to the Linked List using concurrency (multiple goroutines). The key of each node should have a value between 1 to 1 000 000. See the important note below.

Important: Make each goroutine add a subset of nodes - e.g if you use four goroutines every goroutine should try to add 2500 out of the total 10 000 nodes. If you use one goroutine it should add all of them.

Verify that the size (number of nodes) of your Linked List is *as you expect it to be.*

Briefly describe your implementation and the findings from your experiments in the report.

A.2.2 Timing experiment

Lets increase the time it takes for the experiment to complete. Each goroutine should still add only a subset of the nodes. See the important note in 2.1.

a) **Random values**. Record the time it takes for adding a *large number* of nodes with *random values* to the Linked List. Each node should have a *random* value in the range between 1 to 1 000 000. Run the experiment several times - first using one (1) goroutine up to a total of *32 times the number of cores* goroutines in your computer. *Again*: each goroutine should only add a subset of nodes, see 2.1. See the notes on cores above. See the notes on how to find a large number below.

b) Create a diagram where you plot the timing result as a curve based on the number of goroutines. Show the *number of goroutines* on the X-axis and the *time* it took to add all nodes on the Y-axis.

c) **Number sequence**. Re-run the timing experiment, but this time do not use randomly numbered key-values. Instead, generate values that follow a strictly increasing evenly spaced number sequence (still between the range 1 to 1 000 000). Add the result from the experiment to your diagram. Compare the results from the two experiments (random numbers vs. number sequence).

Describe your experiment, the findings and your diagrams in the report.

Notes:

Strictly increasing evenly spaced number sequence:

For example: [1, 11, 21, 31...].

"*large number*":

To find out how many a "*large number*" of nodes is, try to design your experiment so that it takes roughly 10 seconds to run experiment 2.2 with only one (1) goroutine on your computer. Ensure that subsequent re-runs take approximately the same amount of time.

On my machine I found that a *large number* of nodes was 100 000 (with key-values between 1 to 1 000 000). It took 13 seconds to add the nodes to a *course-grained* Linked List using one goroutine. This number of nodes is your baseline for task 2.2 and onward.

You do *not* have to implement or hook-up a plotting pipeline tied to go with some kind of diagram/graphing package. You can simply copy your data and use a program such as e.g Libreoffice/Google Calc/Excel to draw your diagrams.

You can use time.Duration() for timing. Ensure you add the timing code as close to your Add()-loop as possible.

A.2.3 Fine-grained locking

Implement a thread-safe *fine-grained locking* Linked List solution (*hand-over-hand* locking as described in the lecture).

a) Re-run the experiments as described in A.2.2 using this approach.

b) Create an additional diagram showing these results.

Compare the coarse-/fine-grained results and reason on your findings in the report - based on the concepts discussed in the course.

A.2.3 Additional diagrams

- Coarse- *and* Fine-grained locking: with *only* Random number curves
- Coarse- *and* Fine-grained locking: with *only* Number sequence curves
- All 4 curves in the same diagram

Problem A.3 - Optimistic locking/atomic pointers

Optimistic locking

In addition to solving Problem A.1 and Problem A.2, implement a Linked List based on *optimistic locking* and *atomic pointers* (as described in the lecture).

a) Re-run the same experiment as in A.2.2 using this approach.

b) Add the results of to your existing diagrams.

c) Disable / enable the locking and see / describe what happens.

Compare the results of all three approaches and reason on your findings.

Part B - Sorting algorithms

Problem B.1 - Required for C+

Implement a concurrent Quicksort. Use the structure discussed in the lecture on concurrent algorithms, but replace the shared queue with channels.

Describe your implementation (focused on the concurrency) in the report. Can you beat a serial Quicksort?

Hand-in

Submit your solutions as a single zip file via Moodle.

Please include the details of your machine in the report (e.g. OS, CPU details, number of cores, RAM).

This is a group assignment that can be done in groups of one or two students. Your submission should contain well-structured and organized Go code for the problems with a README.txt (or .md) file describing how to compile and run the Go programs and a PDF-report describing your experiment and findings.