NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Concurrency and Parallelism 2022-23

Lab 08: Multiple Synchronisation Strategies in a Concurrent Hash Map

Hervé Paulino & Alex Davidson

May 2023

Abstract

With this lab assignment you shall understand the trade-offs between the costs and benefits of using different locking strategies.

1 Introduction

In this project, you are given a running Java application, available at the following link:

`https://classroom.github.com/a/BkraTcnd`.

This application launches multiple threads — the number of threads is specified as the second command line argument — operating over a shared data structure (a hash map with collision lists). The supported operations are inserting, removing, and looking up for elements. After a certain time — specified in milliseconds as the first command line argument — the application prints some statistics, and terminates.

You may import the project to Eclipse and work from there. If you prefer to work from the command line, go to the application "root directory" (you shall have three subdirectories: "bin", "src", and "resources") and simply run `make`.

To run the application, execute the command below, where the first argument is the time the application will run (in milliseconds), and the second is the number of threads. The output of such execution is typeset in afterwards.

```
java -cp bin cp/articlerep/Main 1000 1
```

See the example output below:

```
Starting application...
Total operations: 4644330
Total successful puts: 585344 (13%)
Total successful removes: 575931 (12%)
Total successful gets: 1151131 (25%)
Throughput: 4644330 ops/sec
Finished application.
```

2 Lab Work

2.1 Improve your knowledge of the program behaviour

1. Change the program to accept a few more command line arguments that may be relevant for the testing process. For example: size of the hash map and workload characterisation (% insert and % remove, the % lookup may be inferred).

2. Change the program to print some additional output information. For example: the min, max, and average size of the collision lists, and the percentage of empty/non-empty collision lists.

2.2 Add automatic validation

Around line 68 of the `Main.java` file there is a *sanity check*, which implements some basic verification of the final state of the concurrent hash map. However, this validation is too weak and passing this validation does not mean that there were no concurrency conflicts.

Add a more elaborated automatic validation to your program. For this you must:

1. **think** about the invariants of the hash map and of the application;

2. **identify** which of those invariants may be broken when running the application with multiple threads;

3. **change** the current *sanity check* to implement checks for the identified invariants. Depending on the additional invariants you plan to check, it may be necessary to change the interface and the implementation of the hash map;

> **Before you start fixing the concurrency issues in the hash map, be sure that you develop a set of strong and solid tests, and that your program fails those tests frequently.**

2.3 Implement different locking strategies to fix the concurrency issues

Create some different implementations of the hash map, each one using a different locking strategy:

1. Create a solution that implements a **coarse-grained locking strategy** using the **Java construct** `synchronised`. With this implementation, there is a single global lock, and only one thread at a time will be allowed to access the hash map.

2. Create a solution that implements a **coarse-grained locking strategy** using a **single plain (exclusive) lock** from the `java.util.concurrent.locks` package. With this implementation there is also a single global lock and only one thread at a time will be allowed to access the hash map.

3. Create a solution that implements a **coarse-grained locking strategy** using a **single read-write lock** from the `java.util.concurrent.locks` package. With this implementation, many lookup operations (`get`) may access the hash map simultaneously, but update operations (`put` and `remove`) must access it one at a time.

4. Create a solution that implements a **medium-grained locking strategy** (i.e., one lock for each collision list of the hash map) using **plain (exclusive) locks** from the `java.util.concurrent.locks` package. With this implementation, many operations may access the hash map simultaneously, as long as they are operating on different collision lists.

5. Create another solution that implements a **medium-grained locking strategy** (i.e., one lock for each collision list of the hash map), this one using **read-write locks** from the `java.util.concurrent.locks` package. With this implementation, many operations may access the hash map simultaneously as long as they are operating on different collision lists, or they are multiple `get` operations on the same collision list.

6. ***OPTIONAL:*** *Create a solution that implements a* ***fine-grained locking strategy****, namely* ***hand-over-hand, optimistic, and/or lazy sychronisation****, to control the concurrent accesses to the hash map collision lists.*

2.4 Evaluate the effectiveness of each locking strategy

Run tests to evaluate the correctness of each of your solutions from above.

If and only if your implementations prove correct, then run more benchmarks and performance tests to evaluate the scalability of the application when you increase the number of threads. Run experiments with $1, 2, 4, , N$ threads, where N is double the number of *hardware threads* you have available in your computer.

NOTE: if you are using some kind of virtualised environment, make sure that you have more than one processor assigned to your virtual execution environment. Otherwise, you will have no speed-ups with more than one thread.

2.5 Acquire a deeper understand of your Java program

Install the `visualvm` (`https://visualvm.github.io/features.html`) tool with the following command (or download from the link provided):

```
apt install visualvm
```

and then use these tools to analyse the multiple versions of the concurrent hash map.

2.6 To Think More About...

The sequential (unprotected) version is much faster than any lock-based solution. Will any of your solutions ever exhibit a speed-up instead of a slowdown? How many processors would you need for that?