departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism 2019-20

Multiple Synchronization Strategies in a Concurrent Article Repository

João M. Lourenço

May 19, 2020

Abstract

With this lab work you will learn about (and practice) correctness of concurrent programs, program invariants, synchronising Java threads; and validating (testing and debugging) concurrent programs.

1 Objectives

In this project you are provided with a running Java application downloadable from BitBucket ("`https://bitbucket.org/joaomlourenco/article-rep.git`"). The application accept a command line argument to control the number of threads. The application is correct only when executed single-threaded. Appropriate synchronisation must be added for the application to work correctly in a multi-threading setting.

The application simulates a repository of scientific articles. Each article has a title, a set of co-authors and a set of keywords. The application uses three hash maps to implement an "in-memory database": one hash-map to keep the articles' information (`byArticleId`), another hash-map to map authors to articles (`byAuthor`), and a third hash-map to map keywords to articles (`byKeyword`). These hash maps are defined as:

```
private Map<Integer, Article> byArticleId;
private Map<String, List<Article>> byAuthor;
private Map<String, List<Article>> byKeyword;
```

where the hashmap `byArticleId` maps integers (the *article identifier*) to articles; the hashmap `byAuthor` maps author names to a list of paper IDs authored by that author; and the hashmap `byKeyword` maps keywords to a list of paper IDs assigned with that keyword.

The article constructor is defined as:

```
public Article(int id, String name) {
    this.id = id;
    this.name = name;
    this.authors = new LinkedList<String>();
    this.keywords = new LinkedList<String>();
}
```

where the `id` is a unique identifier for the paper, the `name` is the title of the paper, and both `authors` and `keywords` are lists with the author names and keywords respectively.

The behaviour of the application is configurable by arguments in the command line (9 obligatory arguments). The application prints some statistics when terminating.

It is possible to enable periodic consistency self-checking in the application by setting a property from the command line by adding

```
-Dcp.articlerep.validate=true
```

right after the "java" in the command line. When the self-checking is active, every 5 seconds all the application threads are "suspended" and a validation thread runs a set of self-checking tests. If no problems are found the application threads are resumed, otherwise an error message is print and the application aborts.

Why may the consistency self-checking fail?

The paper ID is used in the `byAuthor` and `byKeyword` hashmaps as an external key to reference the papers from that author or containing that keyword. When there are concurrent inserts and removes, although the hashmaps individually are each ok, the contents of the three hashmaps are not, e.g., it may happen that the `byAuthor` hashmap is referencing a paper ID which was already removed from the `byArticleId` hashmap. Think... how can this happen? ;)

2 Compiling and Running

To compile the application, go to the application root directory (you shall have three subdirectories: "bin", "src", and "resources") and type the command:

```
make
```

To run the application from the same (compilation) directory, run the command below. It will print some info on the required command line arguments.

```
java -cp bin cp/articlerep/MainRep
usage: cp.articlerep.MainRep time(sec) nthreads nkeys put(%) del(%) get(%)
       nauthors nkeywords nfindlist
```

The command line arguments are:

Argument	Explanation
time	for how long will the program run (in seconds).
nthreads	the number of threads that will operate concurrently on the "in-memory database".
nkeys	size of the hash-map (smaller size implies more collisions).
put	percentage of insert operations.
del	percentage of remove operations.
get	percentage of lookup operations (put + del + get = 100).
nauthors	average number of co-authors for each article (higher number implies more conflicts).
nkeywords	average number of keywords for each article (higher number implies more conflicts).
nfindlist	number of authors (or keywords) in the find list (read carefully the explanation of the *get* operation below).

The lookup (*get*) operation works as follows:

- Half the times (50% of the *get* operations) will do a lookup by authors. The other half will do a lookup by keywords.
- Generate a set A with *nfindlist* authors/keywords (selected randomly from the universe of authors/keywords).
- For each author/keyword i in the set A, create a set P_i with the articles containing i as a co-author/keyword.
- Create the set P of all articles verifying the lookup condition ($P = \cup P_i$).

Higher values in the *nfindlist* argument implies higher contention in the accesses to the "in-memory database".

Example of a possible command line:

```
java -Dcp.articlerep.validate=true -cp bin cp/articlerep/MainRep 10 4 1000 10 10 80 100 100 5
```

You should try with other values to acquire some sensitiveness to the effect of each parameter.

3 Workplan / Methodology

3.1 Add more invariants to the automatic validation

Study the behaviour of the data worker and the way the hash-maps are used in the application. Can you devise some additional invariants that are not being checked?

If you do, you may add your new invariants to the periodic checking phase or as a final checking, jut before the application prints the statistics.

3.2 Add appropriate synchronisation to the program

Add appropriate synchronisation to the program. You should not change the source code logic to avoid the concurrency errors!

You must add synchronisation mechanisms in the appropriate locations for the code to run correctly. If for adding the appropriate synchronisation mechanisms you need to do small changes in the code, e.g., create new private methods, add new public methods; change the arguments of the existing methods, etc.

I suggest you make three versions of the solution:

1. **Global lock (all three hash maps).** Implement a solution using a global lock that will block the accesses to the three hash maps at the same time. This solution shall work correctly by shall not scale (and the global lock eliminates all the concurrency).

2. **Global lock per hash map.** Implement a solution using a global lock per hash map. Although this will ensure that each hash map individually will be consistent, the global contents of the three hash maps may be inconsistent, e.g., one hash map references an Article that is missing in the other hash maps.

3. **Multiple locks per hash map (at the collision lists)**. This solution is harder to implement, but it shall provede both correctness and performance, exhibiting some speedup when the number of processors increase.

3.3 Evaluate the effectiveness of your synchronisation strategy

Run tests to evaluate the scalability of your solution when you increase the number of threads. Run experiments with $1, 2, 4, \ldots, N$ threads, where N is the double of the number of processors/cores you have available in your development computer.

NOTE: if possible please run your tests in *real hardware*, i.e., if you are using a virtual machine (like VMWare, Virtualbox, etc) as your developing environment please copy your files to one of the computers in the lab and run the tests there.

Document History

Version	Date	Description
1.1	2020-05-19	Clarification of the expected work in Section 3.2.
1.0	2020-05-19	Initial version.