

Performance Comparison Between Different Concurrent Lists

Claudio Scheer
claudio.scheer@edu.pucrs.br

May 27, 2020

1 Introduction

In this report, I will present performance results using different list approaches. Five approaches were tested: coarse list, fine list, lazy list, lock free list and optimistic list. Each of these concurrent lists has methods for adding items, removing items, and testing whether the list contains a specific item.

To assess performance, I used the implementations provided by the authors of the book *The Art of Multiprocessor Programming*. These implementations needed a few changes discussed in Section 2. As all implementations were in Java, I also executed all tests in Java.

I do not discuss in depth the differences between the lists in this report, as these differences are discussed in detail in Section 9 of the book *The Art of Multiprocessor Programming*. In addition, the differences and how each list is implemented were widely discussed in class.

In doing so, in Section 3, I discussed the methodology used in the tests and, in Section 4, I present the results of the tests.

2 Problems with the authors' implementation

Two main problems were found: an implementation error in the optimistic list and outdated implementations. All implementations were downloaded from this website: <https://booksite.elsevier.com/9780123973375>.

In the provided implementation of the optimistic list (`OptimisticList.java`), authors use the following method to add an item to the list:

```
1 public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4         Entry pred = this.head;
5         Entry curr = pred.next;
6         while (curr.key <= key) {
7             pred = curr;
8             curr = curr.next;
```

```

9      }
10     pred.lock();
11     curr.lock();
12     try {
13         if (validate(pred, curr)) {
14             if (curr.key == key) { // present
15                 return false;
16             } else { // not present
17                 Entry entry = new Entry(item);
18                 entry.next = curr;
19                 pred.next = entry;
20                 return true;
21             }
22         }
23     } finally { // always unlock
24         pred.unlock();
25         curr.unlock();
26     }
27 }
28 }

```

Listing 1: Method add of OptimisticList.java

On lines 7 and 8, the `pred` variable assumes the current node and `curr` assumes the next node. Therefore, if the `=` sign is used on line 6 of Listing 1, the test on line 14 will never be satisfied and the item will always be added to the list. Hence, to solve this problem, I simply removed the `=` sign on line 6.

Another problem I had with the lists was the different implementations between the 2008 and 2012 book edition. In the 2012 edition, some implementations were updated, but the source code provided was not. Therefore, all implementations were updated according to the 2012 book edition.

The next point is not a problem with the list's implementations, but a necessary feature for my tests. The required feature was a method that could return the size of the list. This method was used to test whether the list size was stable during the test period.

Therefore, to avoid adding an overhead to the methods originally implemented by the authors, I create a method that simply looks for the next item from the head node to the tail node. The nodes are counted and the count is returned. All lists follow the same logic. This method has no blocking, which can make the return of this method out of date. However, in my tests, this did not appear to be a problem.

3 Testing methodology

The experiments were run on a computer with the following configuration:

- **Java version:** openjdk 14.0.1
- **OS:** Ubuntu 20.04
- **Core(s) per socket:** 6

- **Thread(s) per core:** 2
- **CPU(s):** 12
- **RAM:** 32GB

Since the computer has 12 CPUs available, I ran the experiments using 2, 4, 6, 8, 10 and 12 threads. For each number of threads, I tested three list sizes: 100 items, 1000 items and 10000 items. These eighteen experiments were executed three time each and the mean and standard deviation were used to analyze the results.

The number of elements in the list must be stable. Therefore, as the number added and removed from the list is random, I generated random numbers using Equation 1, where N is the size of the list.

$$0 \leq x < (N * 2) \quad (1)$$

There are four possible operations to be performed on a list: add, remove, contains and list size. The list size operation is used to collect the size of the list at a specific point. The probability of each operation is 40%, 40%, 19.9999% and 0.0001%, respectively.

In summary, the specific number of threads is started and each thread gets a random operation and performs it. The number of operations performed on each thread is stored and added to other threads at the end of the test.

The warm-up time used was 15 seconds. The test time was 60 seconds. The operations performed were counted only during the test time. To obtain the throughput of each list, I divided the total number of operations performed on all threads by the test time. In the throughput, the test time also includes the time to interrupt and join the threads.

4 Results

5 Conclusions

6 Miscellaneous

All the source code used in this work are available here: <https://github.com/claudioscheer/concurrent-producer-consumer>.