Distributed Systems: Project Report
Optimistic Lock-Based List-Based Set Implementations

Vorobev Mikhail, Stanislau Palyn

November 17 2024

Contents

1 Abstract **1**

2 Implementations **1**
- 2.1 Hand-over-hand locking . . . 1
  - 2.1.1 *add* method . . . 2
  - 2.1.2 *remove* method . . . 2
  - 2.1.3 *contains* method . . . 3
- 2.2 Hand-over-hand optimized . . . 3
  - 2.2.1 Optimized lock implementation . . . 3

3 Correctness proofs **3**
- 3.1 Proof-sketch of hand-over-hand algorithms . . . 3
- 3.2 Optimized lock . . . 4

4 Performance analysis **4**
- 4.1 Parameters . . . 4
- 4.2 Setup . . . 4
- 4.3 Results . . . 5
  - 4.3.1 Fixed update ratio, varying list size . . . 5
  - 4.3.2 Fixed list size, varying update ratio . . . 6
  - 4.3.3 Fixed list size and update ratio, different algorithms . . . 7
  - 4.3.4 Summary . . . 7

5 References **8**

1 Abstract

This document contains performance comparison of four lock-based list-based sorted set implementations in java: coarse-grained locking, hand-over-hand locking, optimized hand-over-hand locking, lazy linked list by Heller et all. Benchmark were performed using *synchrobench*.

2 Implementations

Implementations of coarse-grained locking and lazy linked list were taken unaltered from the *synchrobench* repository. Implementations of hand-over-hand locking and its optimized version are described below.

2.1 Hand-over-hand locking

Hand-over-hand locking is fine-grained locking algorithm which takes at most two locks while operating on the list. Lock on the predecessor is taken before trying to acquire lock on the successor.

2.1.1 *add* method

```java
public boolean addInt(int item) {
    head.lock();
    Node pred = head;
    Node curr = head.next;
    try {
        curr.lock();
        try {
            while (curr.key < item) {
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock();
            }
            if (curr.key == item) {
                return false;
            } else {
                Node node = new Node(item);
                node.next = curr;
                pred.next = node;
                return true;
            }
        } finally { curr.unlock(); }
    } finally { pred.unlock(); }
}
```

Listing 1: hand-over-hand addInt method

2.1.2 *remove* method

```java
public boolean removeInt(int item) {
    head.lock();
    Node pred = head;
    Node curr = head.next;
    try {
        curr.lock();
        try {
            while (curr.key < item) {
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock();
            }
            if (curr.key == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        } finally { curr.unlock(); }
    } finally { pred.unlock(); }
}
```

Listing 2: hand-over-hand removeInt method

2.1.3 *contains* method

```java
public boolean containsInt(int item) {
    head.lock();
    Node pred = head;
    Node curr = head.next;
    try {
        curr.lock();
        try {
            while (curr.key < item) {
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock();
            }
            return curr.key == item;
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}
```

Listing 3: hand-over-hand containsInt method

2.2 Hand-over-hand optimized

Optimized version of the algorithm is based on the observations that at most two processes compete for the lock of any node except the head. So a less sophisticated (and thus presumably more performant) lock implementation can be used for nodes other than head.

Implementation of the *add*, *remove* and *contains* methods are left untouched. Only underlying locks are optimized for non-head nodes.

2.2.1 Optimized lock implementation

Implementation uses single `volatile Boolean isLocked = false` variable. Methods *lock* and *unlock* are defined as follows:

```java
while (isLocked) { Thread.onSpinWait(); }
isLocked = true;
```

Listing 4: hand-over-hand optimized lock trying section

```java
isLocked = false;
```

Listing 5: hand-over-hand optimized lock unlock method

3 Correctness proofs

3.1 Proof-sketch of hand-over-hand algorithms

In the *add* method, the linearization point can be defined when we identify the first element in the list that is greater than or equal to the target value. When a new node is inserted, the *pred* node remains accessible from the start of the list and cannot be deleted, as it is locked during the operation. The next node *curr* is also locked, so after inserting all three nodes still will be in the list.

In the *contains* method, the linearization point can be established at the moment we reach the conclusion whether the element is present or not and before we unlock the locks. New node with target value cannot be inserted, as we lock its predecessor. Existing node with target value cannot be deleted, as we lock current node as well. So at the moment of unlock our result is correct.

By the same logic as in *add*, other concurrent operations will not affect removal of the target node.

The starvation-free property is maintained because the locking mechanism employed ensures that locks are always acquired in a single direction.

3.2 Optimized lock

This lock is used for non-head list elements. For each of these elements, the following property holds: no more than one process can be in the trying section at any time. This is because every process locks both the current and the previous node as it traverses the list.

Safety: Suppose two processes, p_1 and p_2, are in the critical section simultaneously. Without loss of generality, assume p_1 entered the CS first. Since at most one process can be in the *trying section* (TS) at a time, p_2 could only start its TS after p_1 had finished its TS. At this point, the lock's `isLocked` variable is set to `true`, preventing p_2 from entering the CS. This creates a contradiction, as p_2 should not have been able to enter the CS while p_1 was in it. Therefore, mutual exclusion is guaranteed. The restriction for `isLocked` is just to be a safe register, as concurrent read when we set it to true is impossible, and when we set it to false other process can read any value without violating correctness. We use *volatile* keyword, to make sure that when write is completed, all reads actually can read this updated result.

Liveness: the lock is starvation-free, as process in TS will enter after the process in CS has finished, because it will not be contested by any other process.

4 Performance analysis

4.1 Parameters

Benchmarking was done for each algorithm on the following matrix of parameters:

1. Number of threads: 1, 4, 6, 8, 10, 12.
2. Update ratio: 0 (no updates), 10 (0.1 of queries), 100 (all queries).
3. List size: 100, 1000, 10000.

Following parameters were fixed:

1. Duration: 2 seconds.
2. Values range: twice as big as list size (to make expected list size constant during the benchmark).
3. No warmup.

4.2 Setup

Setup used for benchmarking is:

1. CPU:

```
12th Gen Intel(R) Core(TM) i7-12700H.
14 physical cores, 20 threads.
```

2. Java:

```
openjdk version "21.0.4" 2024-07-16
OpenJDK Runtime Environment JBR-21.0.4+8-569.1-jcef
(build 21.0.4+8-b569.1)
OpenJDK 64-Bit Server VM JBR-21.0.4+8-569.1-jcef
(build 21.0.4+8-b569.1, mixed mode, sharing)
```

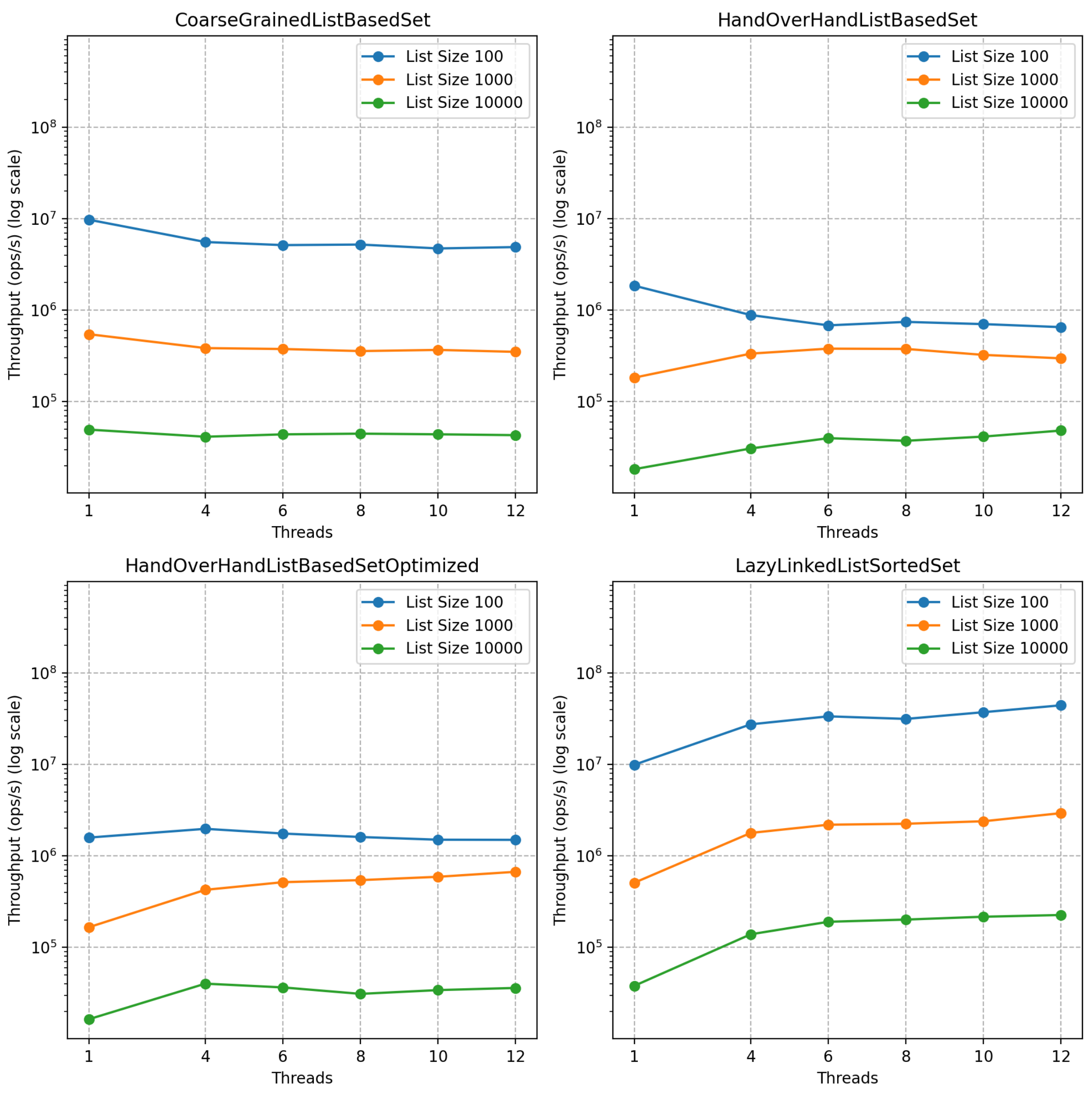

Figure 1: Throughput as function of number of threads for different list sizes. Update ratio is fixed at 10.

4.3 Results

4.3.1 Fixed update ratio, varying list size

Key observations:

- For each algorithm throughput is inversely proportional of the list size. This seems reasonable as with larger lists expected time of an operation for a single thread grows.
- Coarse grained locking does not gain anything from increasing number of threads and performs best with just a single thread. This is expected.
- Both versions of head-over-hand locking do not scale well (at least after 4 threads). Moreover, they perform worse (or maybe just a little better on medium size lists) than coarse grained locking. This is an unexpected result for authors and we struggled to explain this.
- Lazy list performs best and has some scaling.

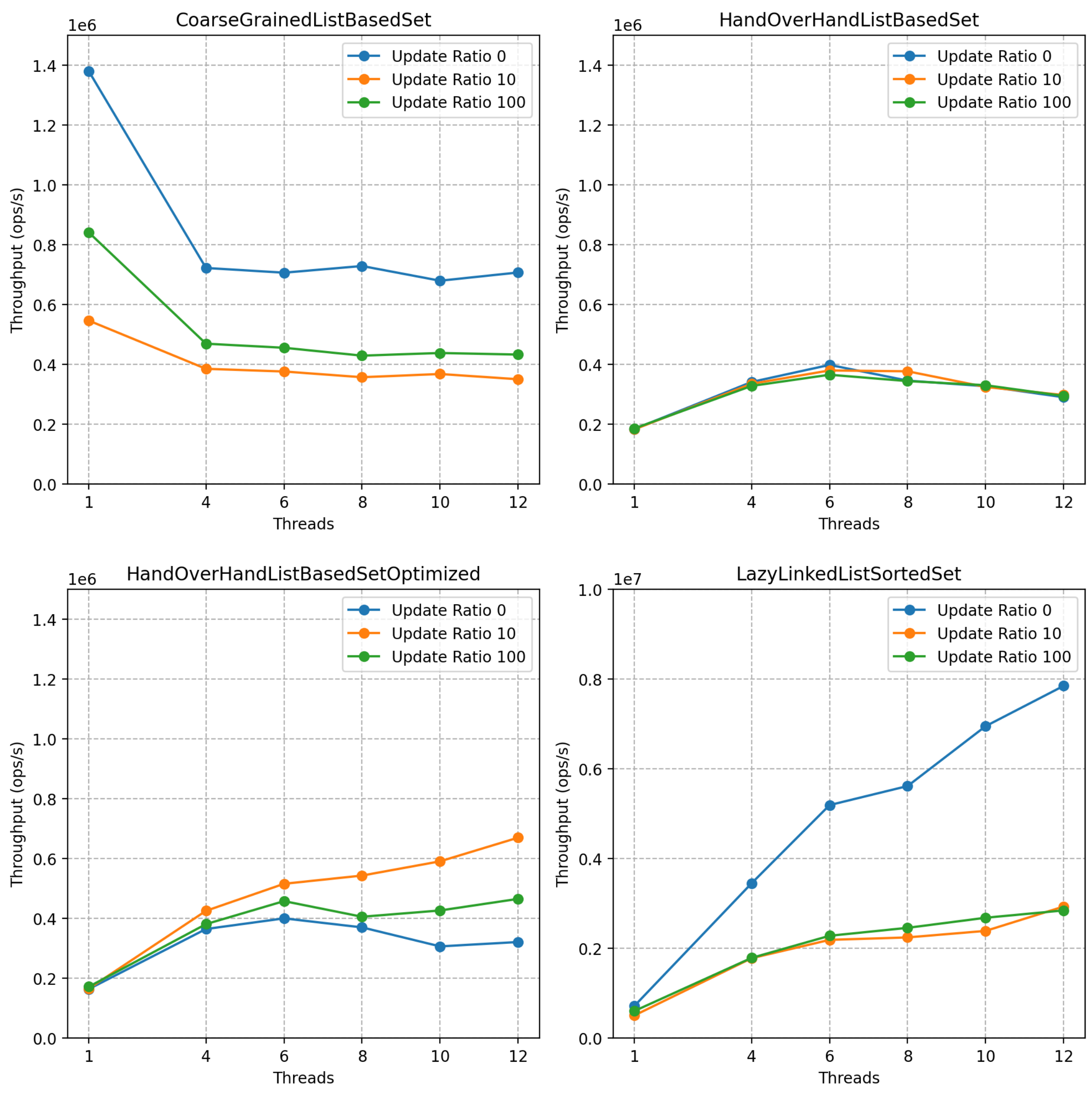

Figure 2: Throughput as function of number of threads for different update ratios. List size is fixed at 1000.

4.3.2 Fixed list size, varying update ratio

Key observations:

- Coarse grained locking performs best with no updates which is expected. However, with only updates it performs better than with 10 percent of them which is strange because containing checks are strictly simpler than updates.
- Both variations of hand-over-hand locking do seem to scale well and have some kind of peak at 6 threads.
- Throughput of hand-over-hand locking does not seem to depend much on update ratio. This is again strange.
- Optimized hand-over-hand performs and scales best with 10 percent of update and shows the worst results with no updates. It is again strange as lookup operations are «lighter» than updates.
- Lazy list shows reasonable scaling and performs best with no updates as expected.

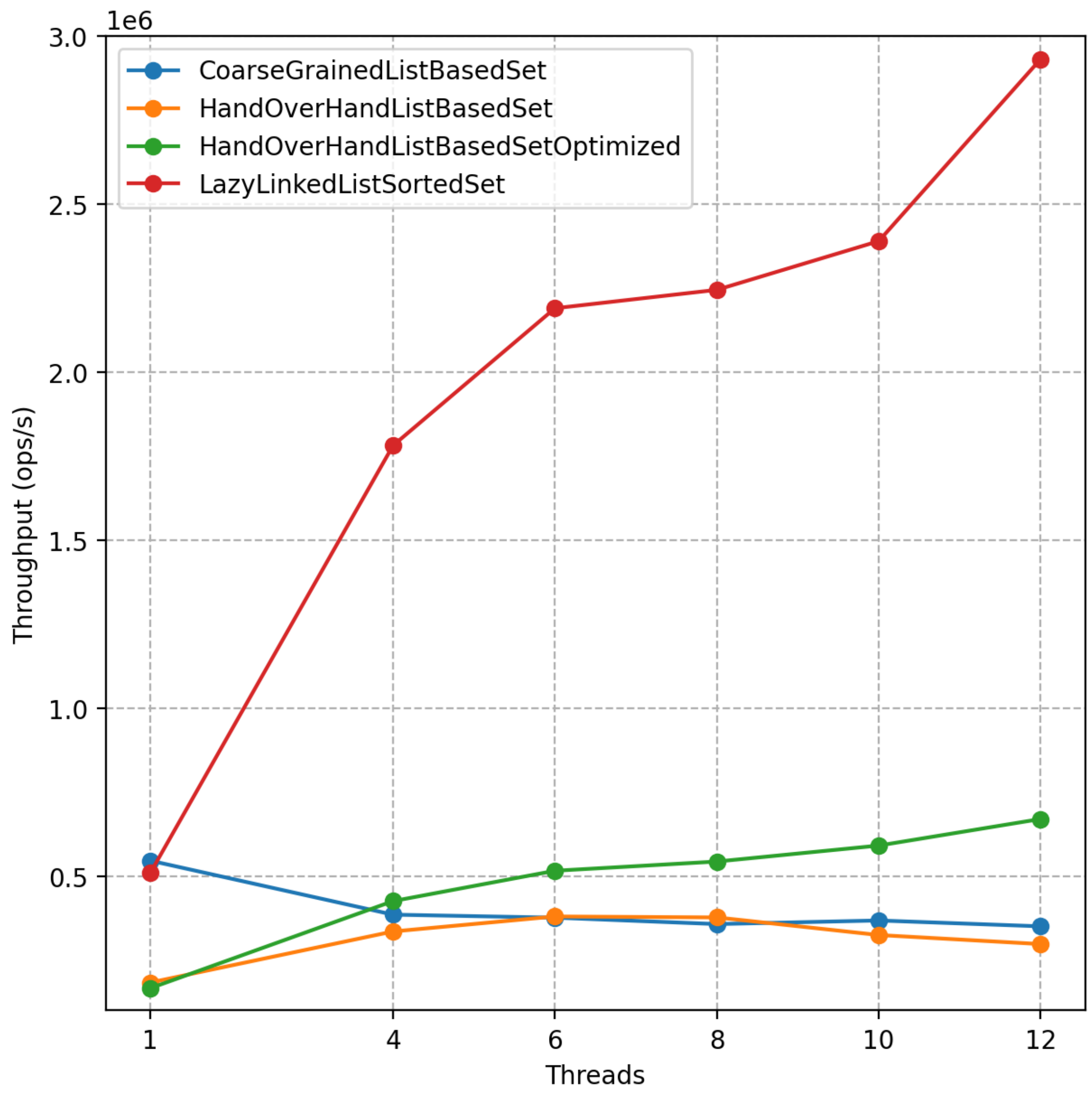

Figure 3: Throughput as function of number of threads for different algorithms. List size is fixed at 1000 and update ratio is fixed at 10.

4.3.3 Fixed list size and update ratio, different algorithms

Key observations:

- Lazy list greatly outperforms other implementations.
- On those parameters optimized hand-over-hand shows small scaling and greatly outperforms unoptimized version.
- However, hand-over-hand locking performs comparably with coarse grained locking. This is unexpected and we struggle to explain this.

4.3.4 Summary

Overall, hand-over-hand locking and its optimized version failed hopes of the authors and didn't show significant performance gain compared to coarse grained locking. The authors are puzzled.

5 References

Project repository can be found at https://github.com/InversionSpaces/ds-cub-project1. Full source code of hand-over-hand locking is at src/linkedlists/lockbased/HandOverHandListBasedSet.java. Optimized version is at src/linkedlists/lockbased/HandOverHandListBasedSetOptimized.java.