SLR206: Project
Optimistic Lock-Based List-Based Set Implementations

Rafael SANDRINI
Benjamin TERNOT

26th October 2022

Contents

I Hands Over Hand algorithm

```java
package linkedlists.lockbased;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import contention.abstractions.AbstractCompositionalIntSet;

/* *
 * SANDRINI Rafael, TERNOT Benjamin
 * Hands over hand lock based algorithm implementation
 * Implementation based on CoarseGrainedListBasedSet, as recommended for
   the project
 */
public class HandsOverHand extends AbstractCompositionalIntSet {

    /* *
     * Node specification, differing from CoarseGrained in an individual
       lock for each node
     * instead of only one node for the entire list
     */
    private class Node {
        private Lock nodeLock = new ReentrantLock();
        public int key;
        public Node next;

        Node(int item) {
            key = item;
            next = null;
        }

        void lock() {
            this.nodeLock.lock();
        }

        void unlock() {
            this.nodeLock.unlock();
        }

    }


    /* *
     * Sentinel nodes
     */
    private Node head;
    private Node tail;

    public HandsOverHand(){
```

```java
        head = new Node(Integer.MIN_VALUE);
        tail = new Node(Integer.MAX_VALUE);
        head.next = tail;
    }

    /* *
     * Insert operation
     * @param item: the element to be added to the list
     * @return: a boolean declaring whether the element was successfully
       added to the list
     */
    @Override
    public boolean addInt(int item){
        head.lock();
        Node pred = head;
        try {
            Node curr = head.next;
            curr.lock();
            try {
                while (curr.key < item){
                    pred.unlock();
                    pred = curr;
                    curr = pred.next;
                    curr.lock();
                }
                if (curr.key==item) {
                    return false;
                }
                Node node = new Node(item);
                node.next = curr;
                pred.next = node;
                return true;
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }



    /* *
     * Remove operation
     * @param item: the element to be removed of the list
     * @return: a boolean declaring whether the element was successfully
       removed of the list
     */
    @Override
```

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

    /* *
     * Contains operation
     * @param item: the element to be checked in the list
     * @return: a boolean declaring whether the element is in the list
     */
    @Override
    public boolean containsInt(int item){
        head.lock();
        Node pred = head;
        try {
            Node curr = head.next;
            curr.lock();
            try {
                while (curr.key < item){
                    pred.unlock();
                    pred = curr;
                    curr = pred.next;
                    curr.lock();
                }
                if (curr.key==item) {
                    return true;
                }
                return false;
```

```
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }

    @Override
    public void clear() {
       head = new Node(Integer.MIN_VALUE);
       head.next = new Node(Integer.MAX_VALUE);
    }

    /**
     * Non atomic and thread-unsafe
     */
    @Override
    public int size() {
        int count = 0;

        Node curr = head.next;
        while (curr.key != Integer.MAX_VALUE) {
            curr = curr.next;
            count++;
        }
        return count;
    }
}
```

II Proof of correctness

II.1 Safety

Safety properties, in a non-formal way, are the properties that ensure that nothing "bad" is ever going to happen during our execution. Linearizability is a safety property that says that, despite concurrency, operations invoked on an object should have linearization points that makes the entire operation appear as a correct sequential execution.

In our Hands Over Hand implementation, we ensure that all the operations occur as a sequential execution, as each operational node is locked by its respective thread at the time of its execution, preventing the other processes from entering the critical section and make any changes together.

II.2 Liveness

Liveness properties, in a non-formal way, are the properties that guarantee that something “good” eventually happens during our execution. Deadlock-freedom is a form of liveness, as there are some processes that will make progress and enter the critical section.

In our Hands Over Hand implementation, the locks are orderly acquired with respect to the node’s key. If one process p cannot acquire its locks, it means that there’s another process q where $q_{key} >= p_{key}$ with the lock, that is, it process has imminent access to critical section, will make progress and unlock its nodes after that.

III Performance analysis

III.1 Fixed update ratio and varying list size

In this section we have one plot per algorithm, with a fixed 10% update ratio, varying the list size in $[100, 1K, 10K]$ and the threads in $[1, 4, 8, 10, 12]$.

III.1.1 Coarse-grained locking algorithm

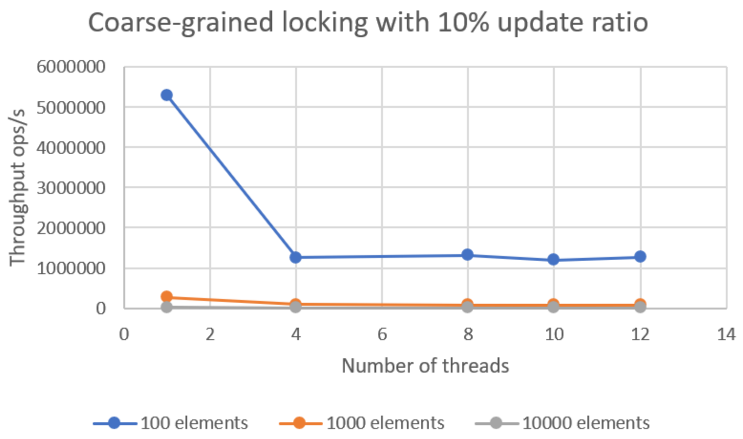

Figure 1: Coarse algorithm

The coarse grained locking algorithm seems to do well with 1 thread and a small list size, but struggle more when there are many threads or a greater list size. Plus, it has a constant speed independent of the number of threads, as soon as there are many.

That might be explained by the fact that with 1 thread, the algorithm does not need to lock anything, but as soon as there are many threads, all is locked, regardless of the greatness of the number of threads. Moreover, with a small list size, the locking time of each operation is smaller because the operations are faster, resulting in a greater throughput.

III.1.2 Lazy linked list by Heller algorithm

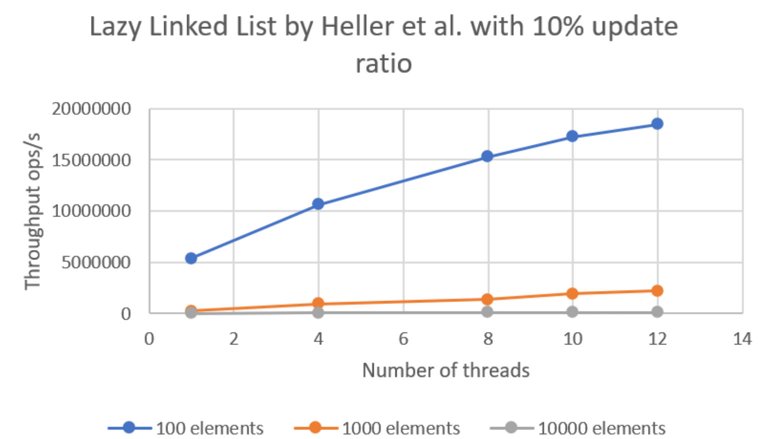

Figure 2: Lazy algorithm

At the contrary of the previous algorithm, the lazy linked list algorithm works better as the number of threads increases. This is because this algorithm is based on a lot of computation, and that with more threads, the computation is more distributed and the efficiency increases.

III.1.3 Hand-Over-Hand locking algorithm

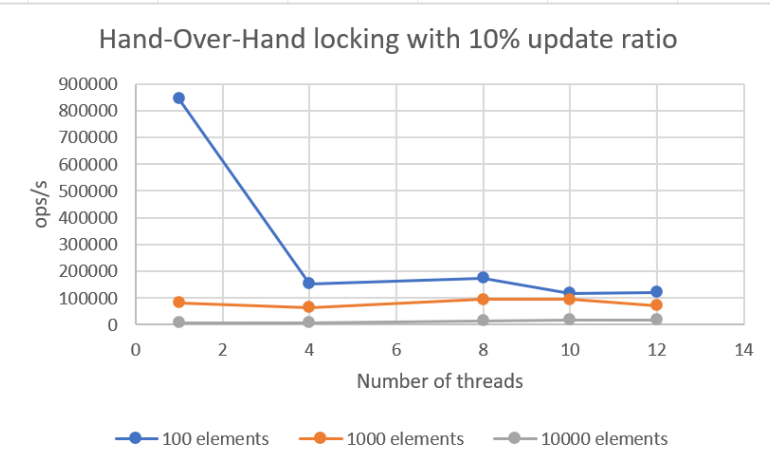

Figure 3: Hand-Over-Hand algorithm

For the hand over hand locking algorithm, except for a small list and 1 thread, the number of threads does not seem to affect the speed. We can check that the behavior of hand-over-hand algorithm is next to the coarse-grained algorithm, it happens because it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive in this approach.

III.2 Fixed list size 100 and varying update ratios

In this section we have one plot per algorithm, with a fixed 100 list size, varying the update ratio in $[0\%, 10\%, 100\%]$ and the threads in $[1, 4, 8, 10, 12]$.

III.2.1 Coarse-grained locking algorithm

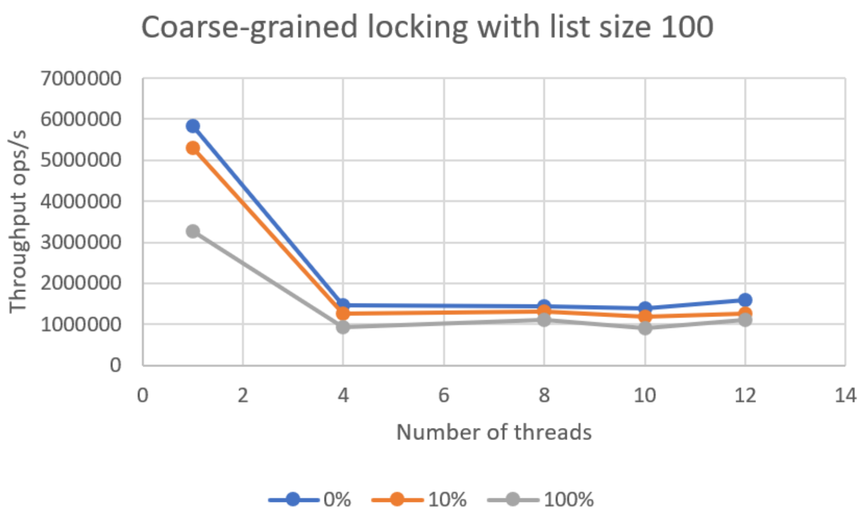

Figure 4: Coarse algorithm

The coarse-grained locking algorithm varies little depending to the update ratio (for the list size 100), but still remains a bit faster when the update ration is small (especially with 1 thread).

III.2.2 Lazy linked list by Heller algorithm

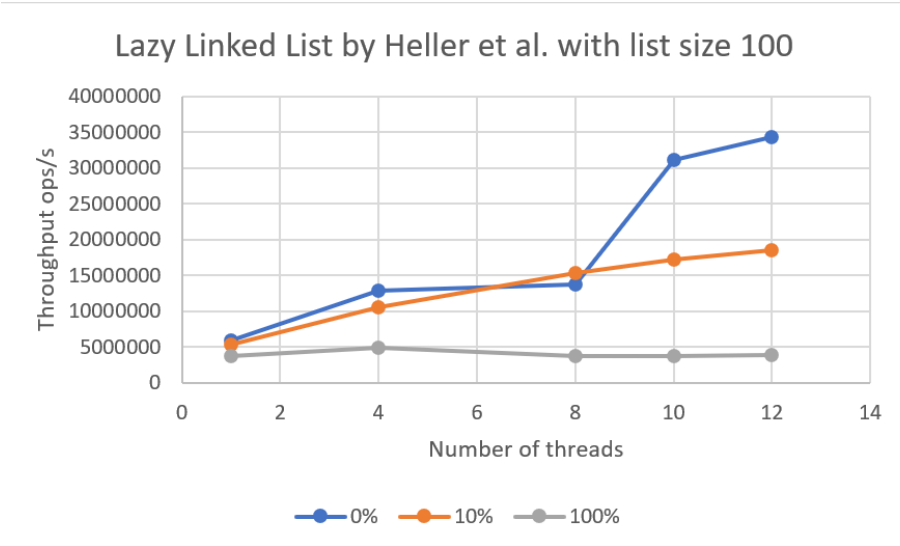

Figure 5: Lazy algorithm

The linked list algorithm depend a lot of the update ratio : it is way more faster with 0% and the speed decreases as the update ratio increases. Plus, the difference of speed between the ratios is greater as the number of threads increases.

III.2.3 Hand-Over-Hand locking algorithm

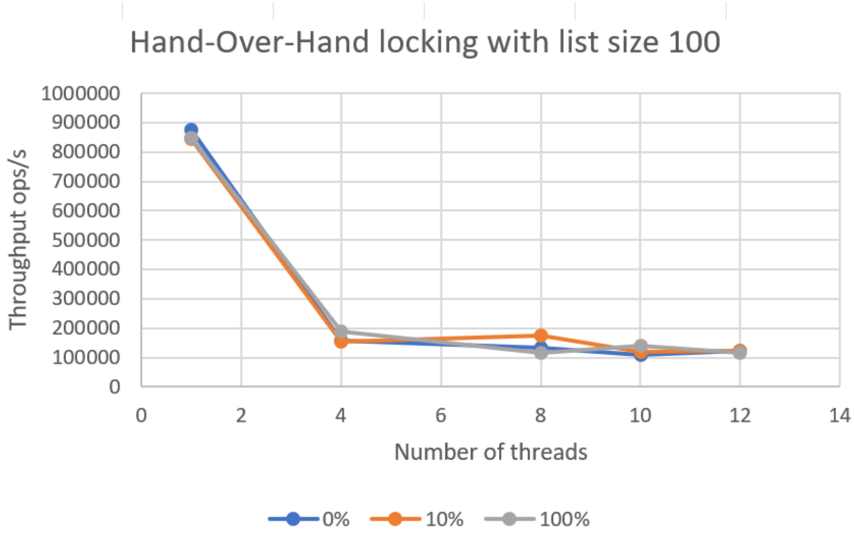

Figure 6: Hand-Over-Hand algorithm

For the last algorithm, we can clearly see that for this list size, the speed seems independent of the update ratio used, and that the only thing that seems to matter is if there is one or many threads.

III.3 Fixed update ratio and list size.

In this section we have one plot with one curve per algorithm, with a fixed 10% update ratio, a fixed 1K list size and varying the threads in $[1, 4, 8, 10, 12]$.

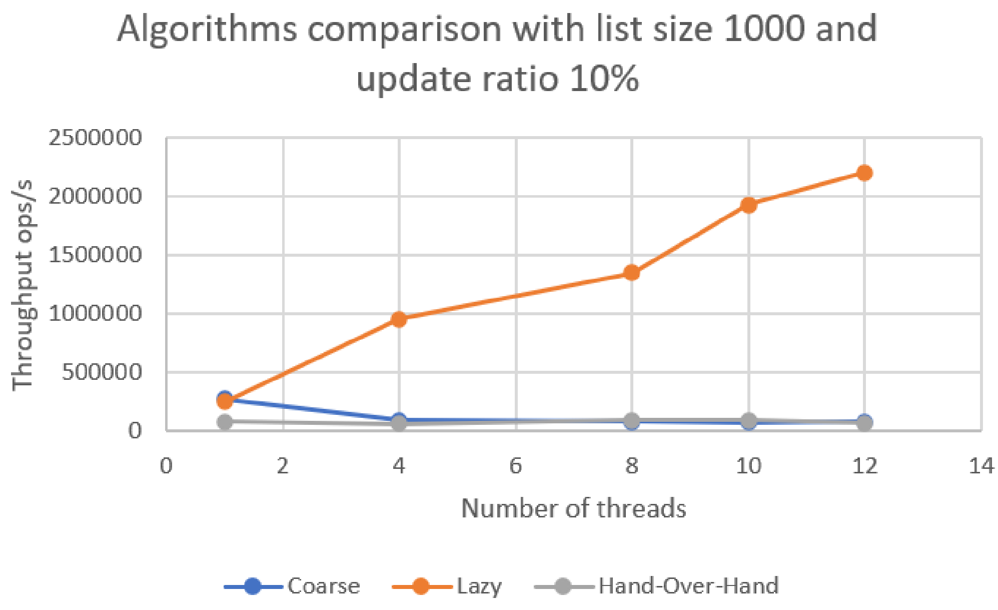

Figure 7: Algorithm performances comparison

Clearly, with theses parameters, the lazy linked lists algorithm is way faster than the two others, and the difference of speed increases with the number of threads, proving that the lazy linked lists algorithm gets a big benefit with concurrency operations. Moreover, as seen in the second experiment, we can also decrease the update ratio and increase more the speed of the lazy algorithm, while the two others won't be affected.

IV System details

```
[bternot-21@lame11]~% lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         46 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  32
  On-line CPU(s) list:   0-31
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz
    CPU family:          6
    Model:               45
    Thread(s) per core:  2
    Core(s) per socket:  8
    Socket(s):           2
    Stepping:            7
    CPU(s) scaling MHz:  41%
    CPU max MHz:         3100.0000
    CPU min MHz:         1200.0000
  NUMA node0 CPU(s):     0-7,16-23
  NUMA node1 CPU(s):     8-15,24-31
Vulnerabilities:
  NUMA node0 CPU(s):     0-7,16-23
  NUMA node1 CPU(s):     8-15,24-31
Vulnerabilities:
  Itlb multihit:         KVM: Mitigation: VMX disabled
  L1tf:                  Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
  Mds:                   Vulnerable: Clear CPU buffers attempted, no microcode; SMT vulnerable
  Meltdown:              Mitigation; PTI
  Mmio stale data:       Unknown: No mitigations
  Retbleed:              Not affected
  Spec store bypass:     Vulnerable
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; Retpolines, STIBP disabled, RSB filling, PBRSB-eIBRS Not affected
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

Figure 8: CPU information of lame11