Report List-based sets

Author: Barau Elena-Raluca, Marculescu Tudor

Table of Contents

Introduction

The goal of this project is to get familiar with optimistic fine-grained locking schemes in concurrent data structures. Our running example is sorted linked list used to implement a set abstraction. The project was conducted in the context of “Fondements des algorithmes répartis (partie A)” course.

It proves that the performance is affected by the synchronization methods used on shared objects. The purpose of this project is to show how different ways of locking a set based list relates to the performance measured in a multi-threaded environment.

The hardware environment used was a virtual machine from the infrastructure of the faculty, namely: **lame21.enst.fr** which has the following specifications:

- Operating System: Ubuntu 24.04.3 LTS

  Kernel: Linux 6.8.0-55-generic
- Processor: 2x Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz

  Architecture: x86-64

  Per processor cores: 10 and threads: 2 per core, in total 20

  Total cores 20, total threads 40
- Memory: 256 GiB
- Storage: HDD 256 GiB

The software environment used to run synchrobench, which is the java based software suite to test the performance of our shared objects is:

- GNU Make v4.3
- Apache Ant v1.9.5 [2]
- Java jdk v1.7.0 [3]
- synchrobench release v1.1.0-alpha [4]

Please notice that Java 8 caused issues when compiling the synchrobench project, particularly when the runtime jar was instrumented to enable transactional memory. We resorted to fallback to Java 7 in order to prevent this issue.

Hand-over-Hand algorithm

The inspiration of implementing this algorithm came from the book “The art of multiprocessor programming” by Maurice Herlihy and Nir Shavit [5]. In order to not start from scratch, the CoarseGrainedListBasedSet file was taken as an example and have been heavily modified in order to introduce the necessary logic of implementation for Hand-over-Hand algorithm.

The methods that have been modified are: addInt(item), removeInt(item) and containsInt(item). Previously, a lock was used in order to prevent the access of the other threads to the whole linked-list of Node objects, which is made obsolete by the Hand-over-Hand algorithm that uses a lock on each Node of the list, thus being an algorithm with a fine grained approach on locking. The Node class was also modified to accommodate this change. To showcase the implementation of the algorithm, it is sufficient to depict the methods of the novel class HandOverHandListIntSet as seen in the figures down below.

```
public boolean addInt(int item) {
       // try to lock the head
       head.lock();
       Node pred = head;
       // proceed to lock the next node
       try {
           Node curr = pred.next;
           // falls asleep until it can lock curr
           curr.lock();
           try {
               // traverse the list until finding the right position
               while (curr.key < item) {
                   pred.unlock();
                   pred = curr;
                   curr = curr.next;
                   curr.lock();
               }
               // if the key is already present, return false
               if (curr.key == item) {
                   return false;
               }
               // else, insert the new node
               Node newNode = new Node(item);
               newNode.next = curr;
               pred.next = newNode;
               return true;
           // unlock curr
           } finally {
               curr.unlock();
           }
       // unlock pred
       } finally {
           pred.unlock();
       }
   }
```

Figure 1 – Hand-over-Hand algorithm insert method

```
public boolean removeInt(int item){
          Node pred = null, curr = null;

        // try to lock the head
        head.lock();
        try {
            pred = head;
            curr = pred.next;

            // try to lock curr
            curr.lock();
            try {
                // traverse the list until finding the right position
                while (curr.key < item) {
                    pred.unlock();
                    pred = curr;
                    curr = curr.next;
                    curr.lock();
                }
            // if the key is present, remove it
            if (curr.key == item) {
                pred.next = curr.next;
                return true;
            }
            // if not found, return false
            return false;
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
```

Figure 2 – Hand-over-Hand algorithm remove method

For both methods, the list is traversed from head to tail by locking 2 successive nodes, the precedent and the current one. In this way, the algorithm allows for a boost in performance if some of the concurrent threads operate on nodes near the tail while others operate on nodes that are near the head, compared to the CoarsedGrained approach in which no matter where the operation of insert or remove took place, if one thread was executing either of them, the others had to busy-wait. By busy-waiting it is meant that the threads do not do any useful work, but waiting to acquire the lock on the linked-list.

The lock() and unlock() methods seen in the figures 1 and 2 are defined in the Node class, that is part of the HandOverHandListIntSet class. They are abstracting the calls to the the lock() and unlock() functions of the ReentrantLock object. Each Node object contains a ReentrantLock object in the case of HandOverHandListIntSet as seen in Figure 3.

```
private class Node {
      Node(int item) {
          key = item;
          next = null;
      }
      public int key;
      public Node next;
      public Lock lock = new ReentrantLock();

      public void lock() {
        lock.lock();
      }

      public void unlock() {
        lock.unlock();
      }
}
```

Figure 3 – Hand-over-Hand algorithm Node of the list

It remains to showcase one final relevant method for synchronization, which is the containsInt(item) method. It is not much different then the removeInt(item) method, but it only searches for the given item in the list, without modifying the Nodes. The method is depicted in the Figure 4.

```
public boolean containsInt(int item){
        head.lock();
        Node pred = head;
        // proceed to lock the next node
        try {
            Node curr = pred.next;
            // falls asleep until it can lock curr
            curr.lock();
            try {
                // traverse the list until finding the right position
                while (curr.key < item) {
                    pred.unlock();
                    pred = curr;
                    curr = curr.next;
                    curr.lock();
                }
                // if the key is already present, return false
                if (curr.key == item) {
                    return true;
                }
                return false;
            // unlock curr
            } finally {
                curr.unlock();
            }
        // unlock pred
        } finally {
            pred.unlock();
        }
    }
```

Figure 4 – Hand-over-Hand algorithm contains method

Correctness of Hand-over-Hand algorithm

In order to prove that the algorithm presented in the previous chapter is correct, it is sufficient to argue that both safety and liveness hold.

Assumptions

Each Node has a lock that provides mutual exclusion. Acquiring a node's lock serializes access to that node and its next pointer.

There are sentinel head (−∞) and tail (+∞) nodes so traversal never sees null.

Invariants to preserve: list is sorted strictly increasing and contains no duplicates; next pointers form a well-formed singly linked list.

Common argument pattern

Show each operation acquires locks in a fixed forward order (lock-coupling): pred then curr, always moving forward. This prevents circular wait → no lock-cycle (so no deadlock).

Show updates to pointers happen only while holding the relevant locks (pred and curr for local updates). Therefore concurrent writers cannot interleave to violate local invariants.

For each operation identify a single atomic linearization point (an atomic memory action performed while holding locks) and argue that this action makes the effect visible to other threads and that it transforms a state satisfying invariants into another state that still satisfies the invariants.

Operation-specific linearization points and safety

addInt(item)

What it does: traverse to locate pred and curr such that pred.key < item ≤ curr.key, with pred and curr locked when the check/insert is done.

Check prevents duplicates: if curr.key == item return false while holding locks.

Linearization point: the assignment pred.next = newNode (performed while pred and curr are locked).

Safety: before the write, pred.next = curr and curr.key ≥ item. After the write, newNode sits between pred and curr, with newNode.key == item and newNode.next = curr, maintaining sorted order and no duplicates (because we checked curr.key != item). No concurrent writer could have changed pred.next or curr while we held pred and curr, so the insertion is atomic at that moment.

removeInt(item) (typical pattern)

What it does: locate pred and curr with pred.key < item ≤ curr.key, with locks held.

If curr.key != item return false.

Linearization point: the assignment pred.next = curr.next (performed while pred and curr are locked).

Safety: removing curr splices it out while holding locks, so no concurrent writer can concurrently splice or insert at the same local place; sorted order remains and duplicates cannot be introduced. If references to the removed node are kept elsewhere, memory reclamation must be handled safely (not shown here).

containsInt(item)

What it does: traverses using lock coupling and tests curr.key.

Linearization point: the read/test of curr.key == item made while both pred and curr are locked (the exact instant of the key comparison).

Safety: because the node(s) read are locked, no concurrent writer can change the relationship (insert or remove in that local region) between the read and the response. Thus the observed membership is consistent with some atomic instant in time.

Argument that these points produce a legal sequential history

For any concurrent execution, replace each operation by its linearization action at the linearization instant. Before that instant the operation had no visible effect; at that instant it performs an atomic pointer/write/read that corresponds to a legal sequential operation (insert/delete/test) on a sorted set; after that it may release locks but further steps do not change the abstract set beyond that linearized effect. Hence each concurrent history is equivalent to some sequential history consistent with operation semantics → linearizability.

Liveness (brief)

Deadlock: avoided because acquisition order is forward along list (no cycles).

Blocking progress: operations block on locks (not lock-free). Under contention throughput degrades; starvation is possible with non-fair locks.

Improvements: make contains optimistic (validate without locks) or use read/write/stamped locks, tryLock+backoff, or lock-free algorithms to improve concurrency and anti-starvation properties

Performance analysis

For the performance analysis, prepare a graph depicting the throughput as a function of the number of threads for the three algorithms, for some representative list size and update ratio. You may use gnuplot or any other plotting program of your choice. Then, for each algorithm, fix the update ratio to 10%, and prepare a graph depicting the throughput as the function of the number of threads, varying the list size. Finally, for each algorithm, prepare a graph depicting the throughput as the function of the number of threads, varying the update ratio, for the list size 1k. Altogether, this gives:

[] Three plots (one per algorithm), with three curves each, for a fixed update ratio 10% and varying list size.

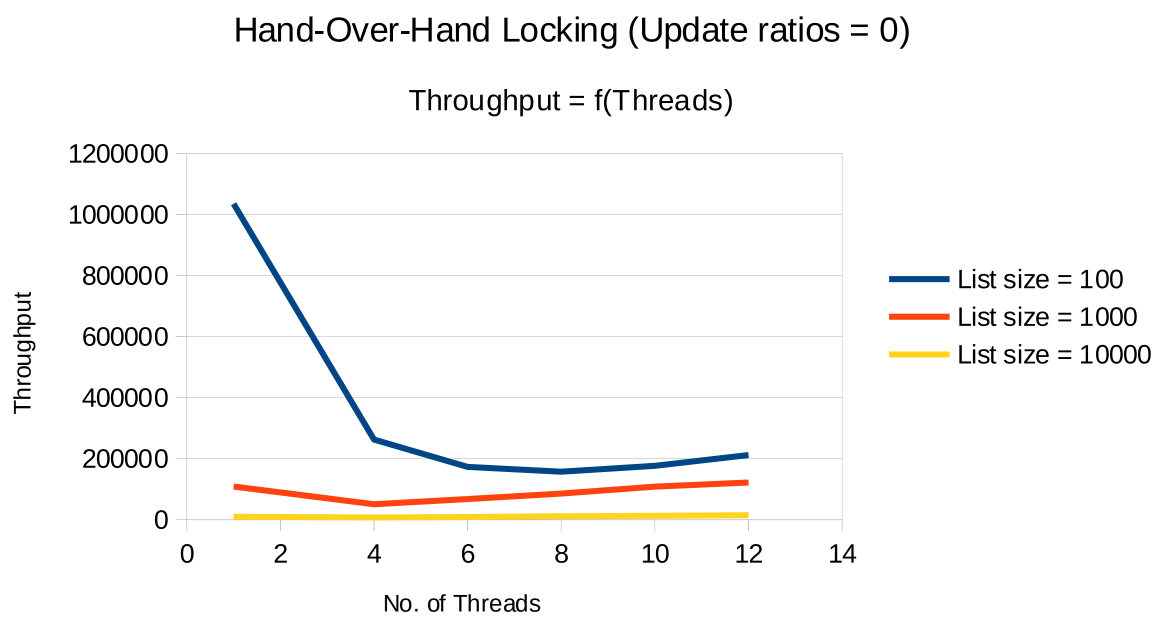

Figure 5: Coarse-Grained Locking algorithm for a fixed update ratio 10% and varying list size

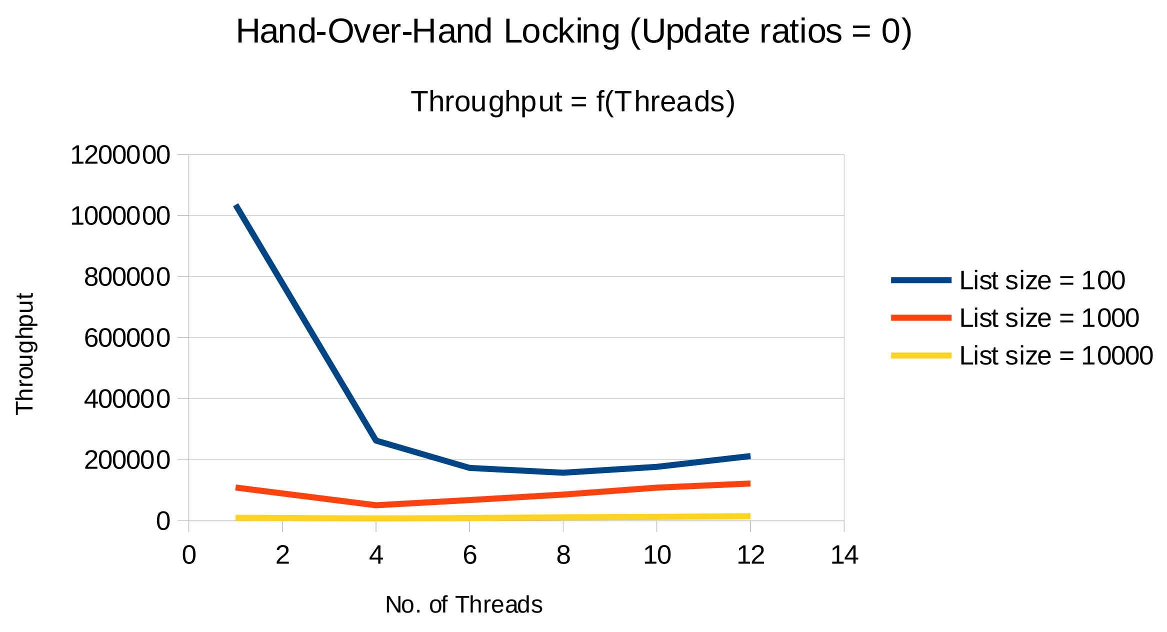

Figure 6: Hand-Over-Hand Locking algorithm for a fixed update ratio 10% and varying list size

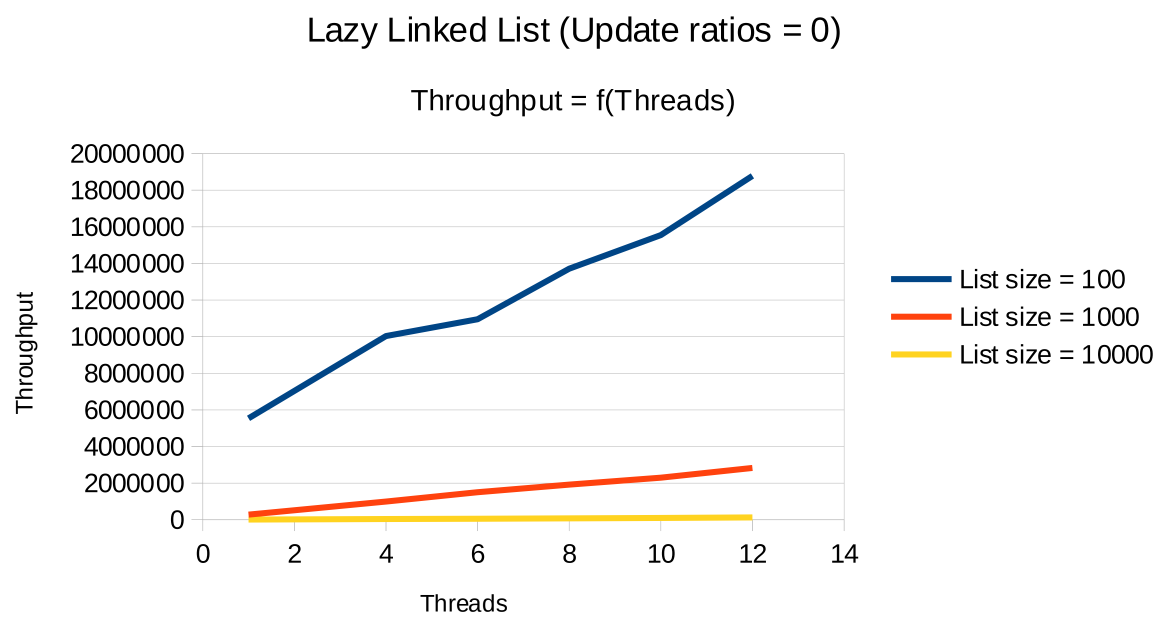

Figure 7: Lazy Linked List algorithm for a fixed update ratio 10% and varying list size

[] Three plots (one per algorithm), with three curves each, for a fixed list size 100 and varying update ratios.

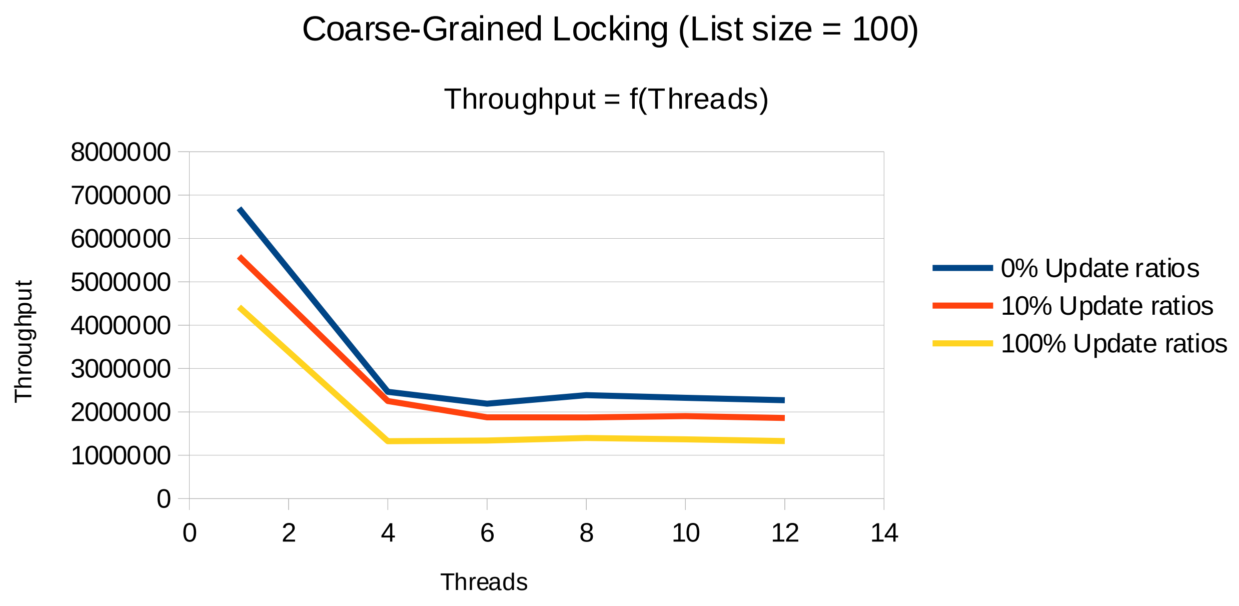

Figure 8: Coarse-Grained Locking algorithm for a fixed list size 100 and varying update ratios

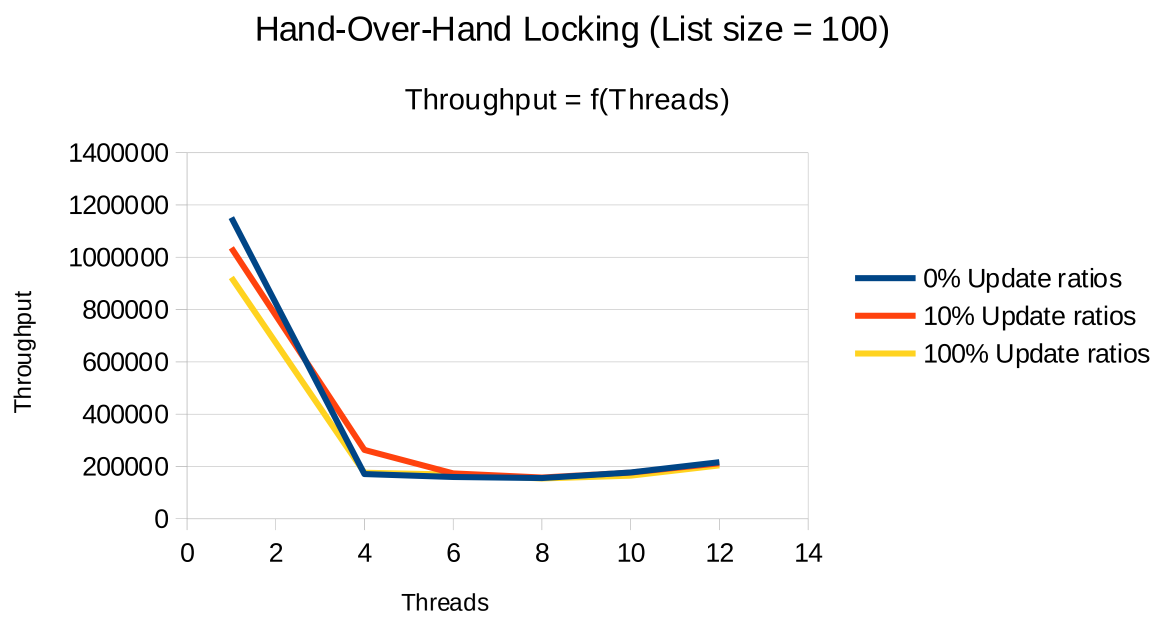

Figure 9: Hand-Over-Hand Locking algorithm for a fixed list size 100 and varying update ratios

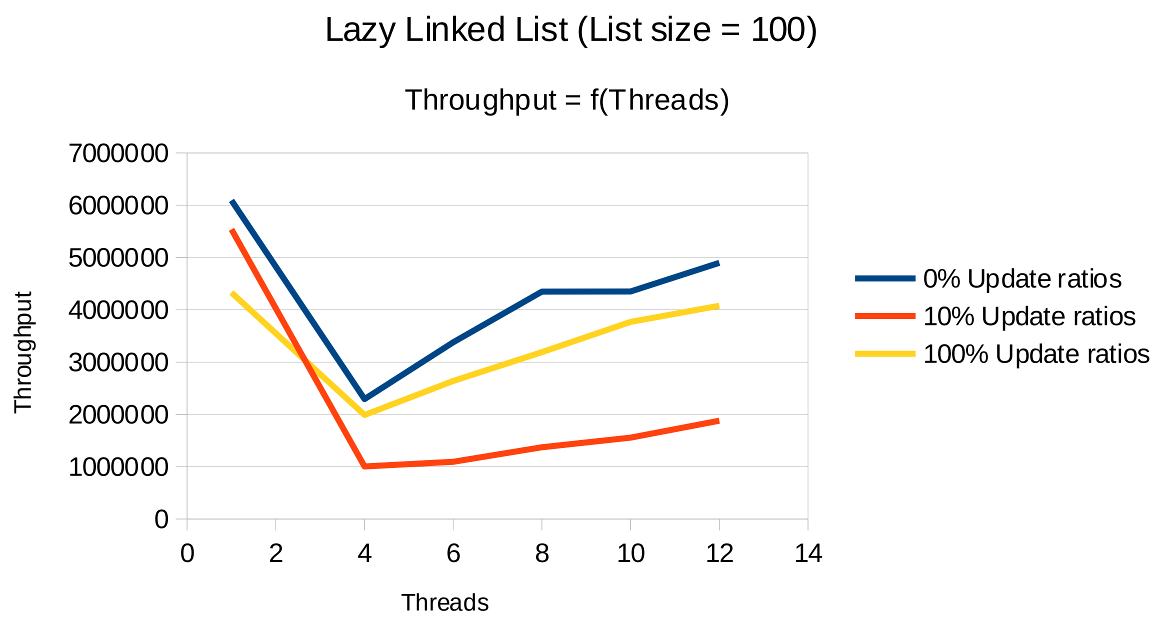

Figure 10: Lazy Linked List algorithm for a fixed list size 100 and varying update ratios

[] One plot, with three curves (one per algorithm), with fixed update ratio 10% and list size 1000.

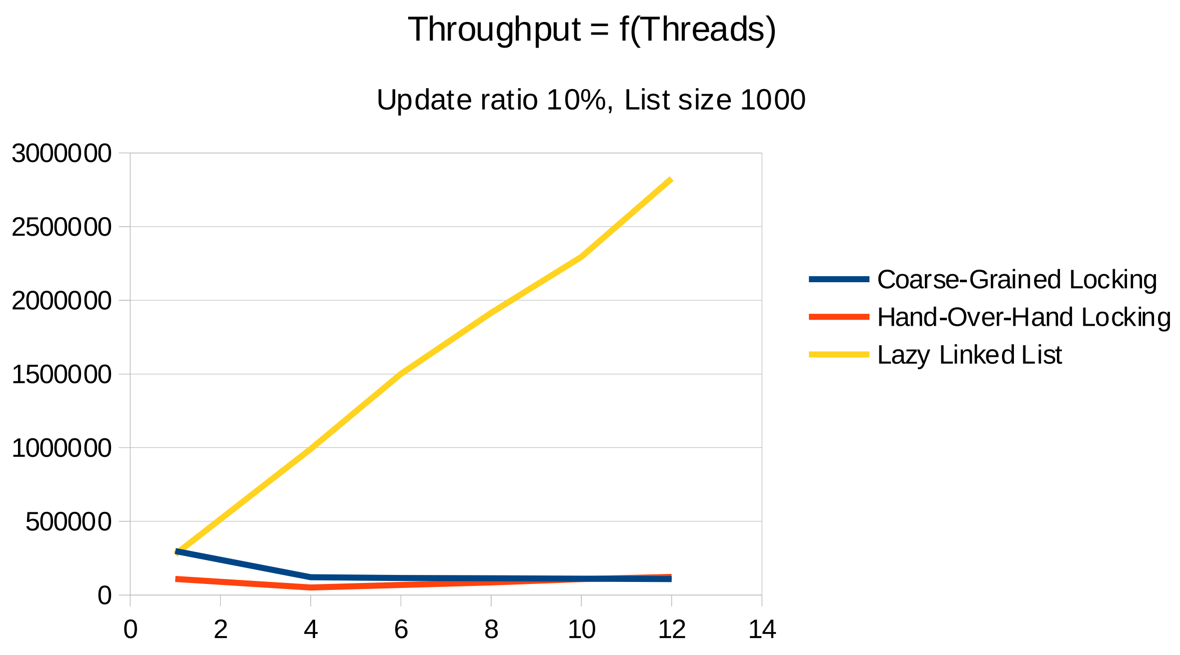

Figure 11: Throughput as the function of the numberof threads for all thre algorithms, with fixed update ratio 10% and list size 1000

[] Explain the forms of the curves and their relations to each other.

Conclusions

In this project, we explored the implementation and analysis of list-based sets using different synchronization strategies in a concurrent environment. By focusing on the Hand-over-Hand (fine-grained) locking algorithm, we demonstrated how carefully managing locks at the node level can significantly reduce contention compared to coarse-grained locking.

The correctness of the Hand-over-Hand algorithm was established through linearizability arguments, showing that all operations maintain the invariants of a sorted, duplicate-free list while preventing deadlocks through ordered lock acquisition. Safety was ensured by holding locks during critical updates, and liveness considerations highlighted the trade-offs between blocking progress and throughput under high contention.

Performance analysis confirmed that fine-grained locking provides better scalability as the number of threads increases, especially for operations distributed across different parts of the list. However, some limitations remain, such as potential contention near the head of the list and the need for fair locks or optimistic approaches to improve throughput further.

Overall, this study illustrates that synchronization strategies have a direct and measurable impact on the efficiency of concurrent data structures. Fine-grained approaches like Hand-over-Hand locking offer a practical balance between correctness, safety, and performance, making them suitable for multi-threaded applications requiring shared access to linked-list-based sets.

References

1. M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, USA: Morgan Kaufmann, 2008

2. Oracle, *Java SE Development Kit 7u80*, v.1.7.0. https://www.oracle.com/in/java/technologies/javase/javase7-archive-downloads.html (last checked: 27.10.2025)

3. Apache, *The Apache ANT Project*, v.1.9.4. https://ant.apache.org/bindownload.cgi (last checked: 27.10.2025)

4. Synchrobench, *A Java-based benchmarking suite for concurrent data structures*, v1.1.0-alpha. https://github.com/gramoli/synchrobench (last checked: 27.10.2025)

5. P. Kuznetsov, *Concurrent List-Based Sets*, CSC_4SL05_TP, 2020