Report List-based sets

Author: Barau Elena-Raluca, Marculescu Tudor

Table of Contents

Introduction

The goal of this project is to get familiar with the 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” course. It proves that the performance is affected by the synchronization methods used on shared objects. Different ways of locking a set based list relates to the performance measured in a multi-threaded environment. Synchrobench, a java based software suite that test the performance of the locking schemes, was used to create the multi-threaded environment and to obtain the throughput as a performance metric. All the code and documents referenced in this report are present on the github repository from the references section. [1]

The algorithms investigated in this report are: CoarseGrainedListBasedSet, HandOverHandListIntSet and LazyLinkedListSortedSet. CoarseGrainedListBasedSet and LazyLinkedListSortedSet are already part of the Synchrobench suite, whereas HandOverHandListIntSet will be implemented and proved to be correct in this report.

In order to allow reproducibility of the results, both the hardware and the software environment are presented. Starting with the hardware environment that was used to conduct the experiments, namely a virtual machine from the infrastructure of the faculty, with the hostname **lame21.enst.fr.** It had 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 DDR4
- Storage: HDD 256 GiB

The software environment software environment needed to run Synchrobench v1.1.0 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 trying to compile the project, particularly when the runtime jar was instrumented to enable transactional memory. We reverted to Java 7 as a fallback to avoid this issue.

Hand-over-Hand algorithm

The inspiration for implementing this algorithm came from the book “The art of multiprocessor programming” by Maurice Herlihy and Nir Shavit [5]. To avoid starting from scratch, the *CoarseGrainedListBasedSet* file was used as a reference and extensively modified to incorporate the required logic for the Hand-over-Hand algorithm implementation.

The methods that needed modifications are: *addInt(item)*, *removeInt(item)* and *containsInt(item)*. Each list class contains a Node class that defines how the node objects from the list are behaving. 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. This is an algorithm with a fine grained approach on locking, therefore the Node class modified to accommodate this change. To showcase the implementation of the algorithm, it is sufficient to depict the methods of the novel class *HandOverHandListIntSet*, 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 add and remove 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 those that are near the head. In contrast 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. They are abstracting the calls to the the *lock()* and *unlock()* functions of the *ReentrantLock* object. Each Node of *HandOverHandListIntSet* object contains a *ReentrantLock* object, which is 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

Finally, the *containsInt(item)* method is quite similar to the *removeInt(item)* method, except that it only searches for the specified item in the list without altering any of 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 [6].

Starting with safety one can do the following 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 (−MAX_INT) and tail (+MAX_INT) nodes so traversal never sees null and that the list is sorted strictly increasing and contains no duplicates; next pointers form a well-formed singly linked list.

A well-formed singly linked list refers to a properly constructed linked list in which: each node contains a valid data element and a reference (or pointer) to the next node; the list has a clear starting point (the head); the last node's next reference correctly points to null (or None), indicating the end of the list; there are no broken links (e.g., missing or invalid next references) and there are no cycles, meaning traversal from the head will eventually terminate.

Starting with the **addInt(item)** method, it is known that is traverses the list to locate pred and curr nodes such that pred.key < item ≤ curr.key, with pred and curr locked when the check/insert is done. It has a check that prevents duplicates: if curr.key == item return false while holding locks, which prevents a concurrent removal of the node.

One can argue that the **linearization point** for this method is the assignment pred.next = newNode, which is performed while pred and curr are locked. 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 the list's sorted order and no duplicates, since the algorithm checks that 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.**

The **removeInt(item)** method locates pred and curr with pred.key < item ≤ curr.key, with locks held. if curr.key is not item, it returns false.

For this operation, the **linearization** point is the assignment pred.next = curr.next, performed while pred and curr are locked. Removing curr splices it out while holding locks, so no concurrent writer can concurrently splice or insert at the same local place. The sorted order remains, since by removing an element from a sorted list, the list still remains sorted, and duplicates cannot be introduced, since this operation only removes items. If references to the removed node are kept elsewhere, memory reclamation must be handled safely, but in this case, it is the job of the garbage collector.

The **containsInt(item)** method traverses using lock coupling and tests curr.key. The linearization point is the test of curr.key == item made while both pred and curr are locked. Because the node read is locked, no concurrent writer can change the relationship by inserting or removing in that local region between the read and the response. Thus the observed membership is consistent with some atomic instant in time.

Now taking all of the above arguments into consideration, one can prove that they produce a legal sequential history because: for any concurrent execution, by replacing 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 write/read that corresponds to a legal sequential operation insert/remove/contains on a sorted set. After that it may release locks but further steps do not change the abstract set beyond that linearized effect. Therefore, each concurrent history is equivalent to some sequential history consistent with operation semantics. This proves that the concurrent histories of the algorithm are linearizable.

Liveness can be proved by arguing that the algorithm is deadlock free. This is a feature of the way locks are taken, from head to tail by each of the methods discussed for safety. They are enough to argue about since *addInt(item)*, *removeInt(item)* and *containsInt(item)* are the only operations executed concurrently at runtime. Figure 9.10, taken from the book "The art of multiprocessor programming" [5] depicts the case of a deadlock effect if the locks would have not been taken from head to tails in the case of all the methods.

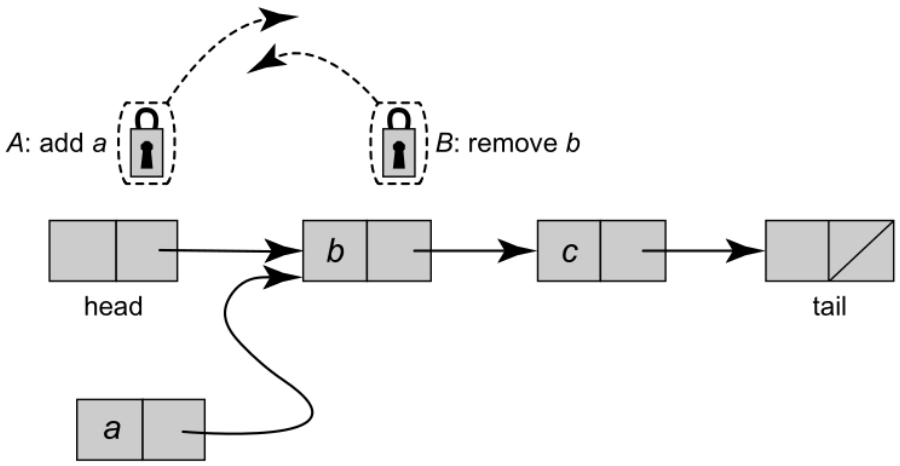

Figure 9.10 The `FineList` class: a deadlock can occur if, for example, `remove()` and `add()` calls acquire locks in opposite order. Thread *A* is about to insert *a* by locking first *b* and then `head`, and thread *B* is about to remove node *b* by locking first `head` and then *b*. Each thread holds the lock the other is waiting to acquire, so neither makes progress.

Performance analysis

For the performance analysis, a graph was created showing the throughput as a function of the number of threads for all three algorithms: CoarseGrainedListBasedSet, HandOverHandListIntSet and LazyLinkedListSortedSet, using representative values for list size and update ratio. Then, for each algorithm, the update ratio was fixed at 10%, and a graph was plotted showing the throughput versus the number of threads while varying the list size. Finally, for each algorithm, a graph was prepared showing the throughput versus the number of threads while varying the update ratio, using a list size of 1k.

In order to measure the performance of the algorithms, the following command was used to call the synchrobench algorithm and obtain the throughput in **operations / second.**

```
java -cp bin contention.benchmark.Test -b linkedlists.lockbased.$OUTPUT -d 2000
-t $i -u $j -i $k -r $r -W 0 | grep Throughput

where:
 $OUTPUT == name of the class containing the algorithm to be used
 -t $i   == sets the number of threads to be used while doing the experiment
 -u $j   == sets the update ratio in percentage while doing the experiment
 -i $k   == sets the initial list size
 -r $r   == sets the maximal list size, the range
 -W 0    == 0 seconds for warmup
 -d 2000 == 2s duration of each experiment
```

In the following figures, the plots are representative for each algorithm with a fixed update ratio of 10% but with varying list sizes.

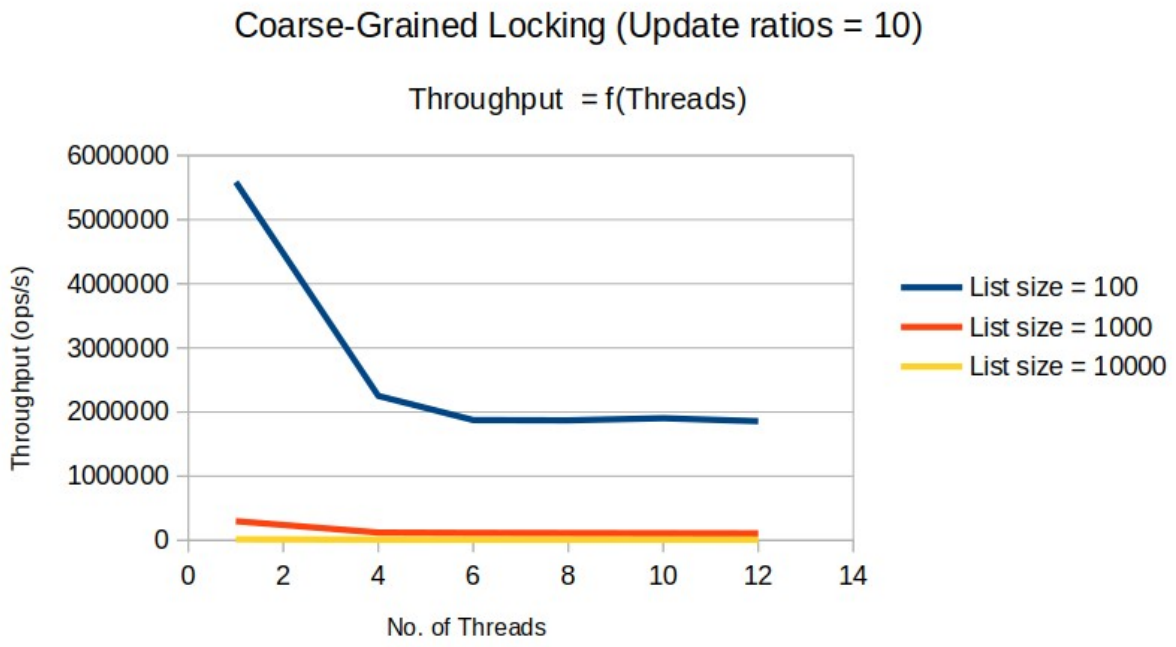

Figure 5: Coarse-Grained Locking algorithm for a fixed update ratio 10% and varying list size

The algorithm's performance on a small list (size=100) reveals a critical weakness: while it achieves the highest single-threaded throughput due to minimal overhead, this speed becomes a liability under concurrency. Threads finish their work so fast that they often try to use the same lock at the same time, which makes throughput drop sharply. In contrast, on larger lists (1k, 10k), the initial throughput is lower, but the performance degradation is more gradual. Because it takes longer to go through a big list, threads don’t all try to get the lock at the same time, which reduces how much they block each other, so the larger list can perform better once a few threads are running.

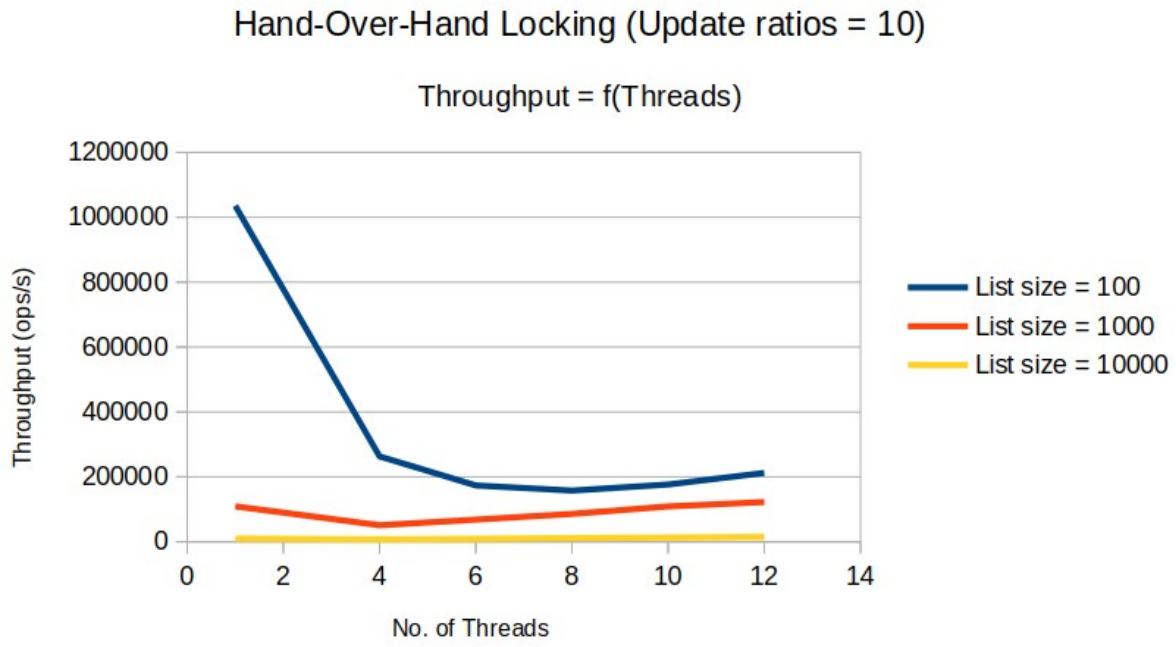

Figure 6: Hand-Over-Hand Locking algorithm for a fixed update ratio 10% and varying list size

A similar trend can be observed in the case of this algorithm, but for a different reason. Comparing the different list sizes shows that the algorithm has a lot of overhead. For the large list (10k), performance is very low and barely changes with more threads. The time spent locking and unlocking so many nodes dominates the runtime and removes any real parallelism. The smaller list (100) runs faster because there are fewer locks, but it still scales poorly. Its sharp drop in throughput shows that even a short list can’t avoid threads blocking each other. The mid-sized list (1k) shows a small improvement, the lock cost is spread out a bit more, allowing for a slight parallel benefit that isn’t seen in the other cases.

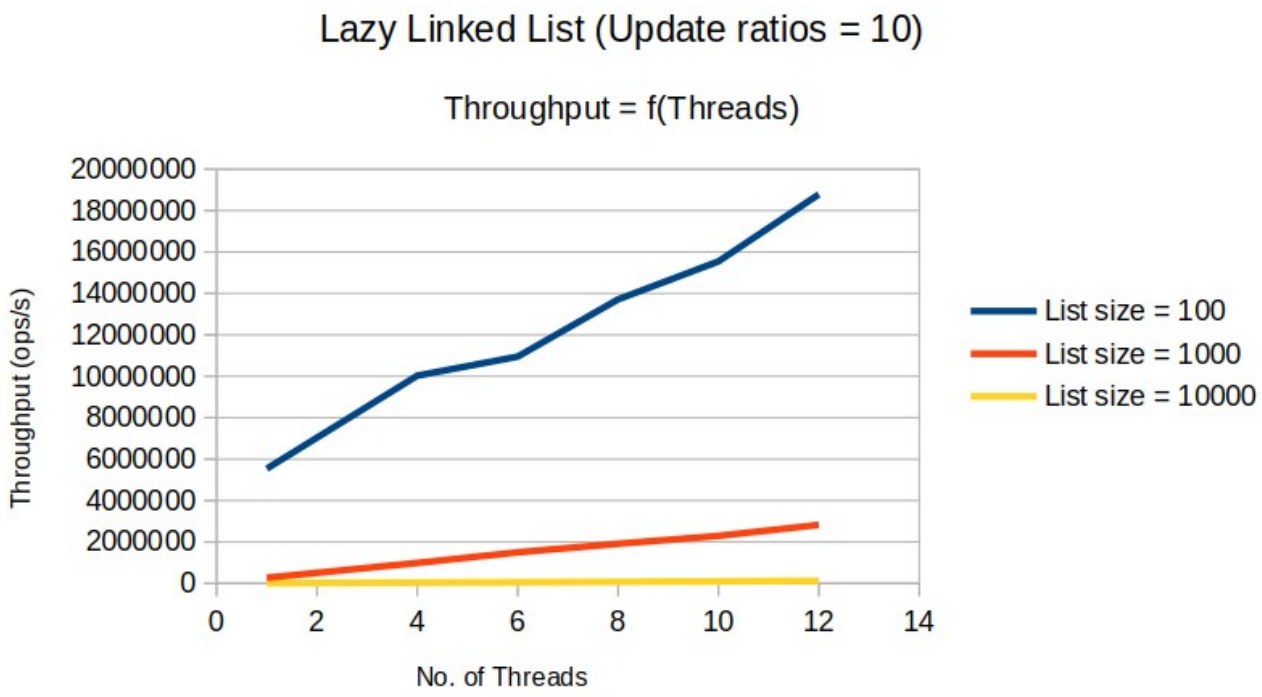

Figure 7: Lazy Linked List algorithm for a fixed update ratio 10% and varying list size

The graph shows that smaller lists enable higher throughput and better scalability, as evidenced by the consistent increase in throughput for all list sizes as the number of threads grows. This confirms the algorithm's effectiveness at leveraging concurrency for its lock-free traversal phase. However, the graph reveals a critical trade-off: the list of size 100 performs best because its short, fast traversals allow threads to complete operations rapidly. The majority of these operations are lock-free reads, minimizing contention. In contrast, the larger lists (1k and 10k) have lower throughput. While long traversals reduce contention by spacing out threads, the computational cost of the traversal itself becomes the new bottleneck, slowing down the overall rate of operations. Therefore, under this read-heavy workload, a smaller list is more efficient than a larger one.

When we change the list size, we can see how much work each algorithm really has to do. Coarse-Grained Locking stays slow no matter the size because all threads fight over one lock. Hand-Over-Hand Locking also struggles — the cost of locking each node is so high that any possible speedup from parallelism is mostly lost. The Lazy Linked List, however, shows clear and consistent scaling. It performs best on smaller lists in this mostly read-based workload, since shorter traversals let it finish operations faster. Overall, the Lazy Linked List is both the most scalable and the most efficient, especially for medium-sized lists.

The following plots depict each algorithm with a fixed list size but varying update ratios.

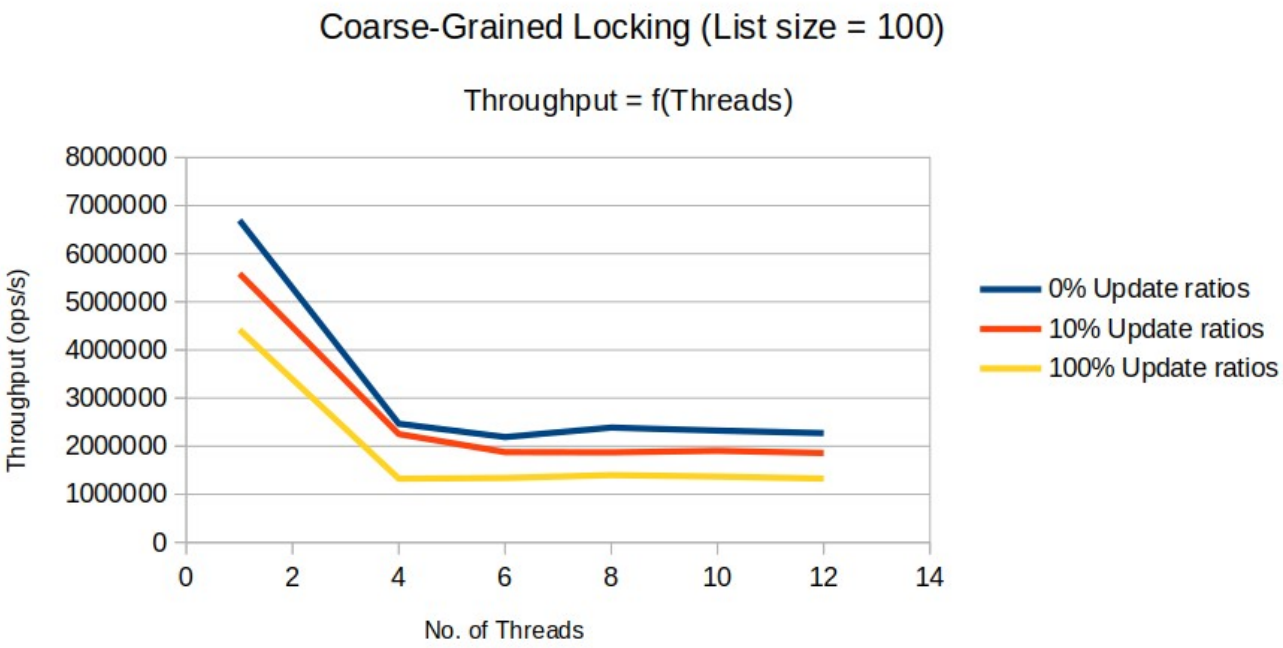

Figure 8: Coarse-Grained Locking algorithm for a fixed list size 100 and varying update ratios

All three curves show the same pattern: a sharp, abrupt drop in throughput between 1 and 4 threads, followed by a nearly constant, low throughput level. The only difference is their vertical position: the 0% curve is the highest, followed by the 10% curve and the 100% curve is the lowest. This ordering is consistent and reflects the cost of operations, reads are cheapest, writes are most expensive due to exclusive access. This is a clear sign of a serious bottleneck: because there's only one global lock, all operations have to take turns and the sharp drop happens right away as threads start fighting for that lock. After that, the system is maxed out, adding more threads just makes them wait longer instead of speeding things up. The gap between the lines shows that even when everything runs one at a time, reads are still faster than writes.

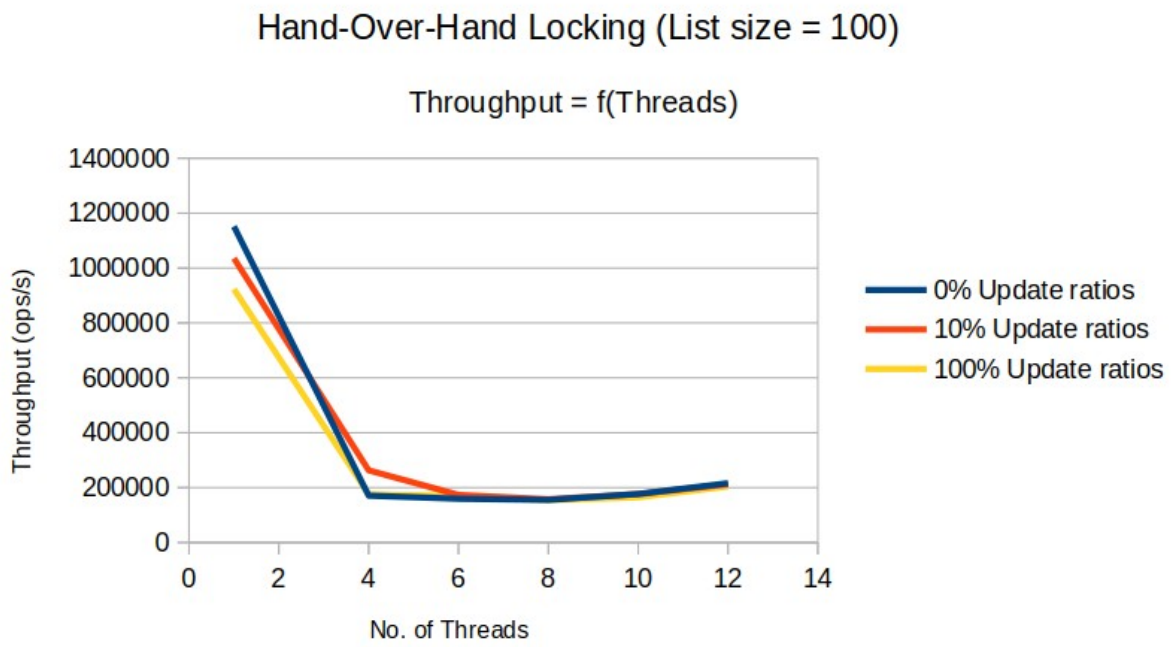

Figure 9: Hand-Over-Hand Locking algorithm for a fixed list size 100 and varying update ratios

The curves for all three update ratios are almost identical, overlapping for most of their range. They all show an initial performance collapse up to 4 threads, followed by a very slow, gradual recovery up to 12 threads. The fact that the 0%, 10%, and 100% update lines are nearly indistinguishable is the most critical observation. This reveals that algorithm's performance is utterly dominated by the cost of locking and unlocking every node it goes through. On a list of size 100, this overhead is so large that it completely masks the fundamental difference between a read and a write operation. The slight recovery at high thread counts may be due to the possibility of some threads occasionally working on different parts of the list at the same time, allowing a bit of real parallelism to show up.

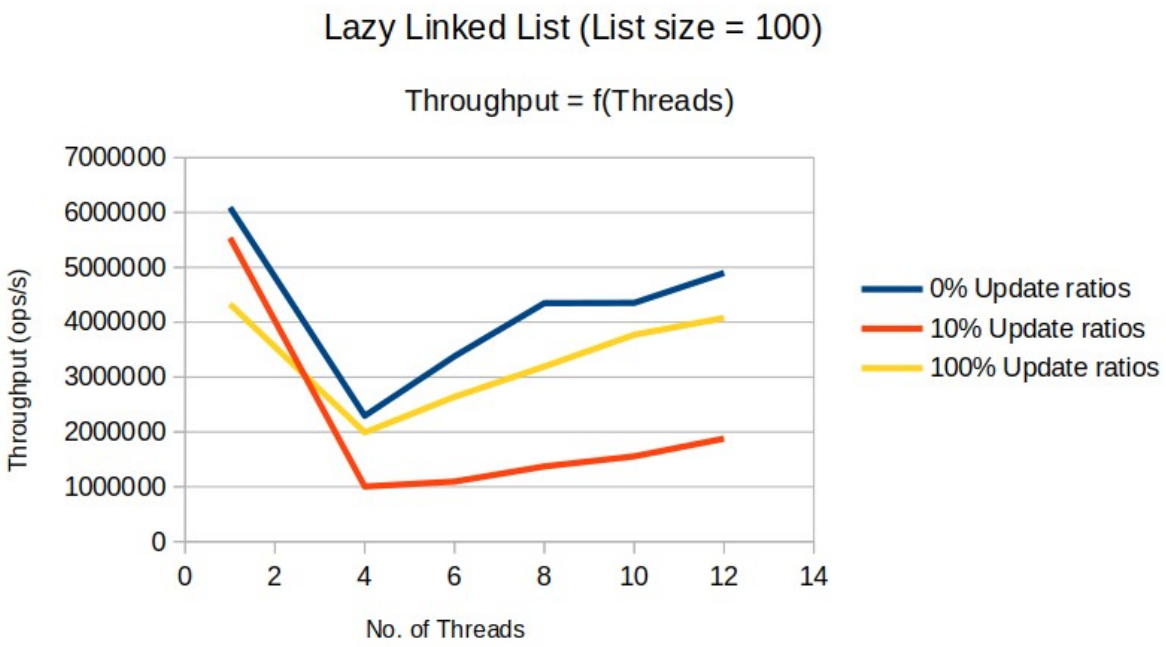

Figure 10: Lazy Linked List algorithm for a fixed list size 100 and varying update ratios

All three curves show a sharp decrease until 4 threads, but then a clear and significant increase in throughput up to 12 threads. The 0% update (read-only) curve starts the highest and stays on top, as expected. What's interesting is that the 100% update (write-only) curve ends up performing better than the 10% update curve after a few threads. For most of the graph, the order is: 0% highest, then 100%, then 10% lowest. The initial drop happens because threads are fighting over the same lock used during updates and validations. The strong recovery shows that the algorithm can actually take advantage of multiple threads once it settles into a balanced workload. The surprising part is when the 100% update curve beats the 10% update one. In the 10% case, most operations are reads, which are fast, but they still have to wait for the occasional write that locks the list. This mix of fast and slow operations causes more conflict. In the 100% case, everything is a write, so even though writes are slower, all threads are doing the same type of work. This makes the system more predictable and smoother overall, reducing the back-and-forth delays seen in the mixed case.

This analysis clearly shows that each algorithm's locking approach determines how it handles different workloads. Coarse-Grained Locking performs poorly in all cases, basically forcing everything to run one at a time. Hand-Over-Hand Locking is also inefficient, since its high per-node locking cost makes it almost ignore the difference between reads and writes. By contrast, the Lazy Linked List scales well and reacts to the workload. It can recover performance with more threads, and even shows surprising behavior, like the write-only workload doing better than the mixed one, because it handles contention smartly. Overall, it's the only algorithm where the performance really reflects what kind of workload it's running.

Finally one plot, with three curves, one for each algorithm, with fixed update ratio 10% and list size 1000.

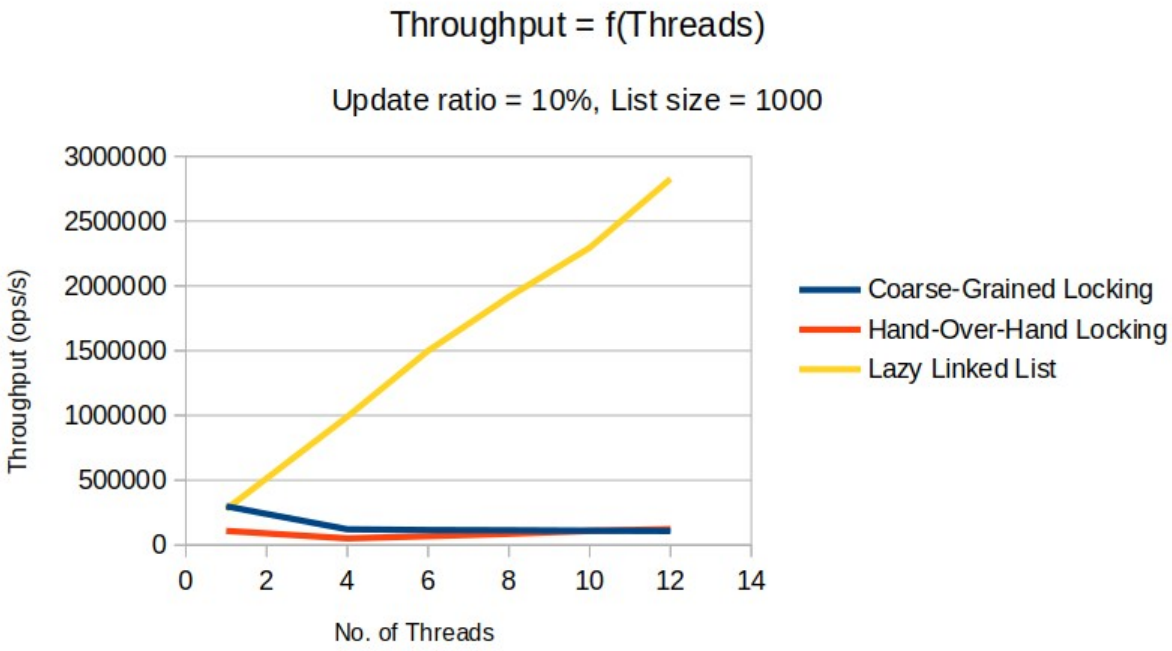

Figure 11: Throughput as the function of the number of threads for all three algorithms, with fixed update ratio 10% and list size 1000

This final graph gives a clear comparison of the three algorithms under a balanced, typical workload, a medium-sized list with a mix of reads and writes. The results clearly show how well each algorithm handles concurrency. The Coarse-Grained Locking algorithm fails to scale. It starts with decent throughput for a single thread, but performance quickly collapses as more threads are added, then flattens out at a very low level. This happens because the single global lock forces all operations to run one at a time, so adding threads actually hurts performance. Hand-Over-Hand Locking performs poorly across the board. It starts with the lowest throughput for a single thread due to high per-node lock overhead, and adding threads only gives a small improvement. While it avoids the complete collapse seen in Coarse-Grained Locking, it's still far behind the best algorithm, showing that the theoretical parallelism is mostly lost to locking overhead. In contrast, the Lazy Linked List scales very well. It starts with high single-threaded performance and keeps increasing steadily as threads are added. This shows it's good at handling concurrency: reads can mostly happen without locks, and updates are done in a way that minimizes contention. For this common workload, it's clearly the best performer.

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 than coarse-grained locking as the number of threads increases, especially for operations distributed across different parts of the list. However, limitations remain, such as contention near the head of the list and the overhead introduced by acquiring and releasing multiple locks. These issues suggest that further improvements require more optimistic or lock-free approaches.

This broader performance analysis clearly shows a ranking among the three concurrent linked list algorithms. Coarse-Grained Locking is limited by contention on a single lock, so performance collapses as soon as multiple threads are added. Hand-Over-Hand Locking is slowed down by its high per-node lock overhead, making it inefficient and not very responsive to different workloads. In contrast, the Lazy Linked List consistently performs the best. It’s the only algorithm that scales effectively with increasing threads, performing especially well in read-heavy workloads while also adapting to mixed workloads. It reaches its best performance on medium-sized lists, where traversal costs remain manageable.

Overall, this study illustrates that synchronization strategies have a direct and measurable impact on the efficiency of concurrent data structures. While Hand-over-Hand locking offers a solid balance between correctness and performance, the Lazy Linked List ultimately provides the best combination of scalability and efficiency, making it the preferred choice for modern multi-threaded applications that rely on shared, concurrent linked lists.

References

1. *optimistic-lock-based-list-based-set,* https://github.com/marcutudor79/optimistic-lock-based-list-based-set (last checked: 27.10.2025)

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

3. 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)

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

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