SLR206 - Foundations of distributed algorithms

Optimistic Lock-Based List-Based Set Implementations

Renan Rodrigues

Hand-Over-Hand Algorithm


```java
package linkedlists.lockbased;

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

/**
 * Implementation of the fine grained lock based set, where it goes
 * through the linkedlist of the set locking only 2 nodes per access.
 */
public class HandsOverHandsSet extends AbstractCompositionalIntSet {
    private Node head;
    private Node tail;

    public HandsOverHandsSet(){
        head = new Node(Integer.MIN_VALUE);
        tail = new Node(Integer.MAX_VALUE);
        head.next = tail;
    }

    /**
     * Add an element to the list.
     * Goes through the list and looking for the place where the element
     * @param x will be inserted locking 2 nodes ad a time. If @param x
     * is already in the set, then it's not added. If the element was
     * added the method returns true, esle it returns false. Once the
     * operation is done, the locks are realesed.
     * @return a boolean indicating if the element were added.
     * @param x the element to add.
     */
    @Override
    public boolean addInt(int x) {
        head.lock();
        Node pred = head;
        try {
            Node curr = pred.next;
            curr.lock();
            try {
                while(curr.key < x){
                    pred.unlock();
                    pred=curr;
                    curr=curr.next;
                    curr.lock();
                }
                if(curr.key == x){
                    return false;
                }
                Node newNode = new Node(x);
                pred.next = newNode;
                newNode.next=curr;
                return true;
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }

    /**
```

```
61      * Remove an element from the list.
62      * Goes through the list and looking for the element @param x to
63      * be removed locking 2 nodes ad a time.If the element was removed
64      * the method returns true, esle it returns false. Once the
65      * operation is done, the locks are realesed.
66      * @return a boolean indicating if the element where found and removed.
67      * @param x The element to remove
68      */
69     @Override
70     public boolean removeInt(int x) {
71         head.lock();
72         Node pred = head;
73         try {
74             Node curr = pred.next;
75             curr.lock();
76             try {
77                 while(curr.key < x){
78                     pred.unlock();
79                     pred=curr;
80                     curr=curr.next;
81                     curr.lock();
82                 }
83                 if(curr.key != x) {
84                     return false;
85                 }
86                 pred.next = curr.next;
87                 return true;
88             } finally {
89                 curr.unlock();
90             }
91         } finally {
92             pred.unlock();
93         }
94     }
95
96     /**
97      * Check whether a element is in the set.
98      * Goes through the list and looking for the place where the
99      * element @param x is located locking 2 nodes ad a time. Once
100     * it finds the place than it returns true if x is in the set,
101     * else it returns false. Once the operation is done, the locks
102     * are realesed.
103     * @return a boolean indicating if the element is in the set
104     * @param x the element to check the presence of
105     */
106    @Override
107    public boolean containsInt(int x) {
108        head.lock();
109        Node pred = head;
110        try {
111            Node curr = pred.next;
112            curr.lock();
113            try {
114                while(curr.key < x){
115                    pred.unlock();
116                    pred=curr;
117                    curr=curr.next;
118                    curr.lock();
119                }
120                return curr.key == x;
```

```
121             } finally {
122                 curr.unlock();
123             }
124         } finally {
125             pred.unlock();
126         }
127     }
128 
129     @Override
130     public int size() {
131         int count = 0;
132 
133         Node curr = head.next;
134         while (curr.key != Integer.MAX_VALUE) {
135             curr = curr.next;
136             count++;
137         }
138         return count;
139     }
140 
141     @Override
142     public void clear() {
143         head = new Node(Integer.MIN_VALUE);
144         head.next = new Node(Integer.MAX_VALUE);
145     }
146 
147     private class Node {
148         public int key;
149         public Node next = null;
150 
151         private Lock lock = new ReentrantLock();
152 
153         Node(int item) {
154             key = item;
155         }
156 
157         void lock(){
158             this.lock.lock();
159         }
160 
161         void unlock(){
162             this.lock.unlock();
163         }
164     }
165 }
166 
```

Proof of correctness

Safety

- **Linearizability**

In a Concurrent or distributed system multiple processes may be accessing a single object, there may arise a situation in which while one process is accessing the object, another process changes its contents. Therefore, it is needed a way, in these kinds of systems, to define which constraints what outputs are possible when an object is accessed by multiple processes at the same time. Linearizability is a safety property which ensures that operations do not complete in an unexpected or unpredictable manner.

In the case of the Hand-over-hand algorithm, It is possible to choose the moment when the nodes were blocked and validated as the linearization point for any operation. Then as with the hand-on-hand algorithm, the consistency and the sequential specification of the set are not violated, since the current node and the predecessor node are locked during the update of the critical section, and since they are validated before the update, no operation can interfere during this process.

Liveness

- **Deadlock freedom**

Liveness implies that a system will make progress despite the fact that its concurrently executing components ("processes") may have to "take turns" in critical sections, parts of the program that cannot be simultaneously run by multiple processes. More generally, a liveness property states that "something good will eventually occur", therefore a liveness property cannot be violated in a finite execution of a distributed system because the "good" event might still occur at some later time.

Based on that Deadlock freedom is a form of liveness, since a system is said to be Deadlock-free when there is a group of processes competing for access to the critical section at some point in time, then some process eventually makes progress at a later point in time.

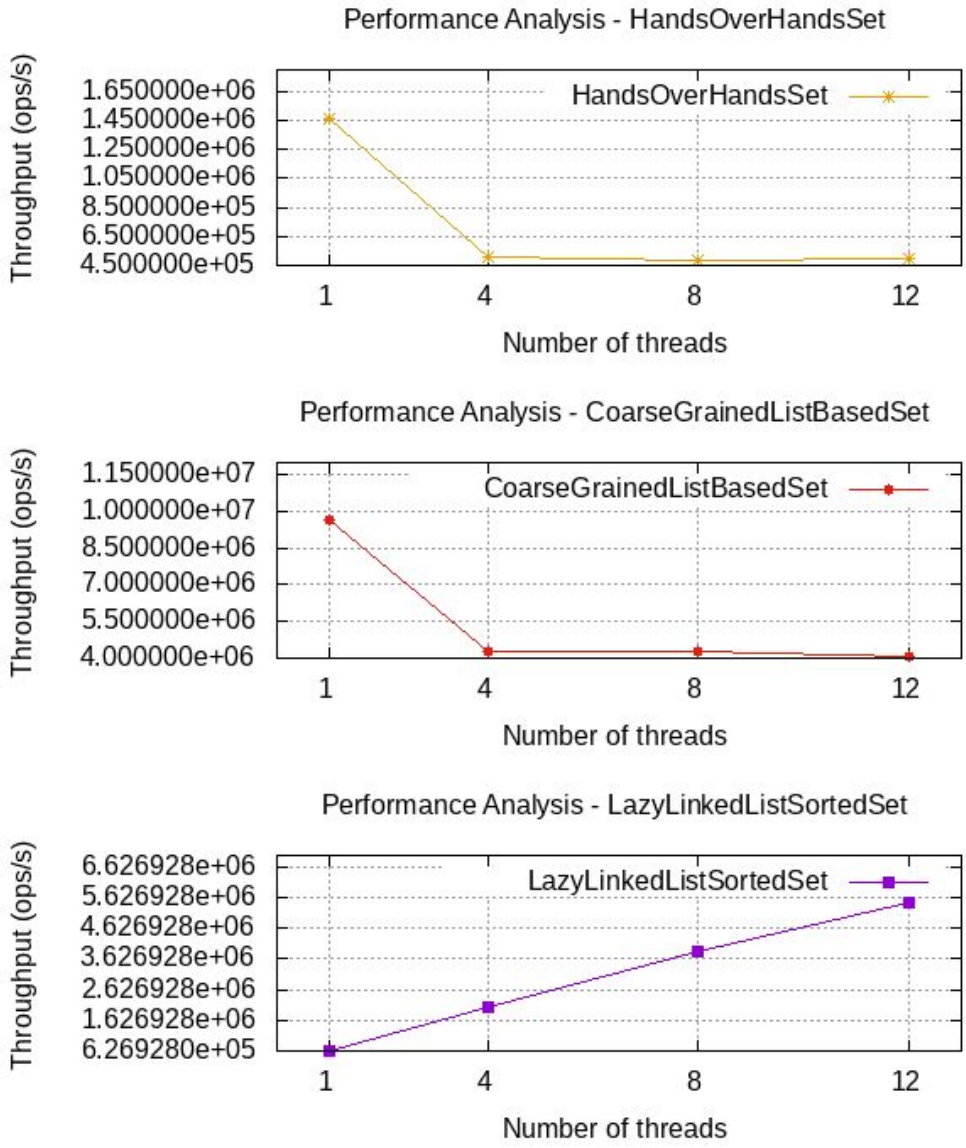
In the case of the Hand-over-hand algorithm the Deadlock freedom is verified, since any process is able to progress during the wait-free traversal step. Thus, if the nodes that were found could not be blocked or validated, it means that another process is or has been able to progress. Thereby, verifying the liveness present in the algorithm.

Performance analysis

In this section, it will be analyzed the graph depicting the throughput as a function of the number of threads for the three following algorithms: CoarseGrainedListBasedSet, HandsOverHandsSet and LazyLinkedListSortedSet. In order to generate the graphs, it was used the program gnuplot.

Algorithm performance comparison

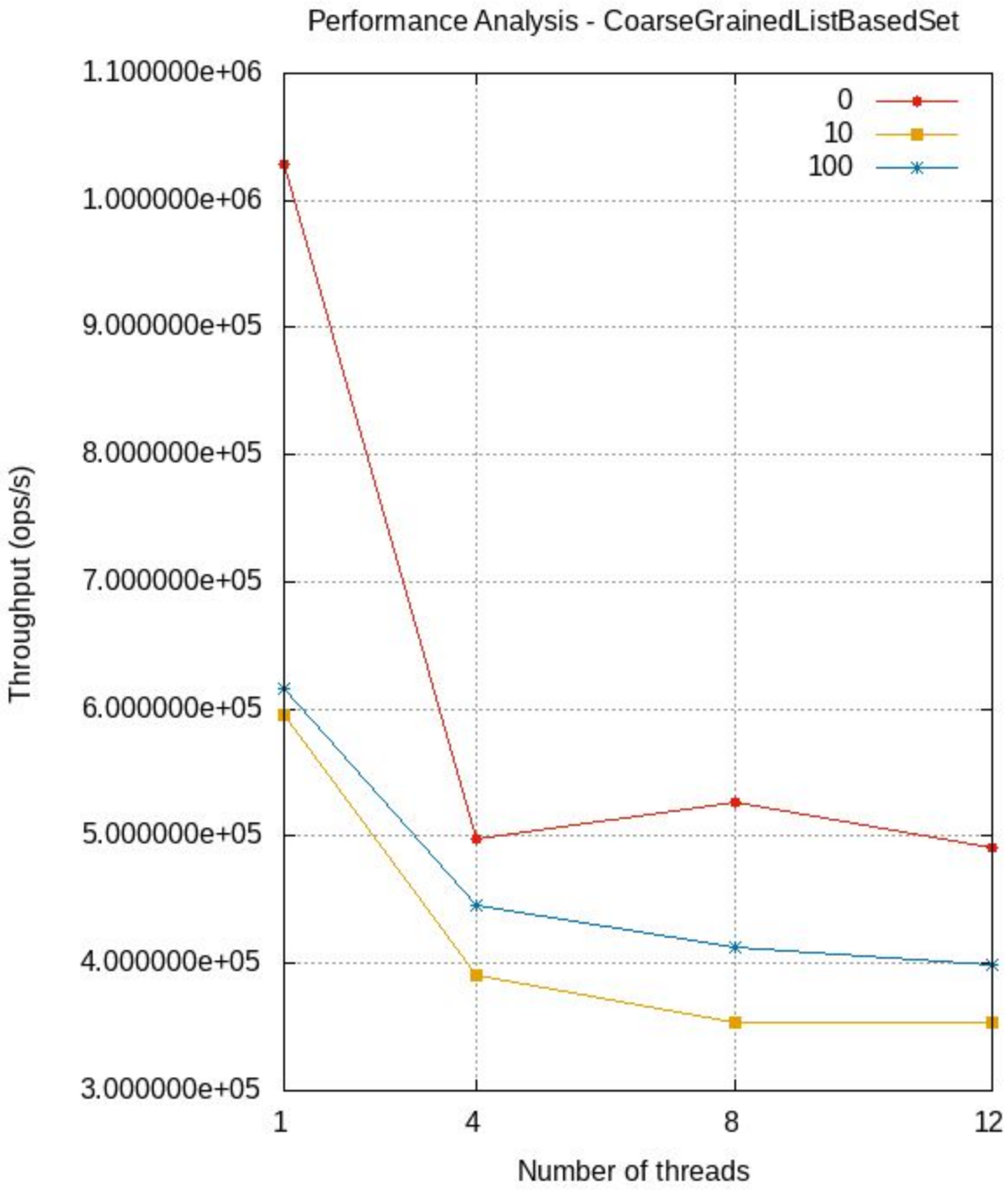
For this first analysis, it was chosen as both a fixed “UPDATE_RATIO” and “LIST_SIZE” for each of the algorithms of respectively, 10 and 100. The plot follows:



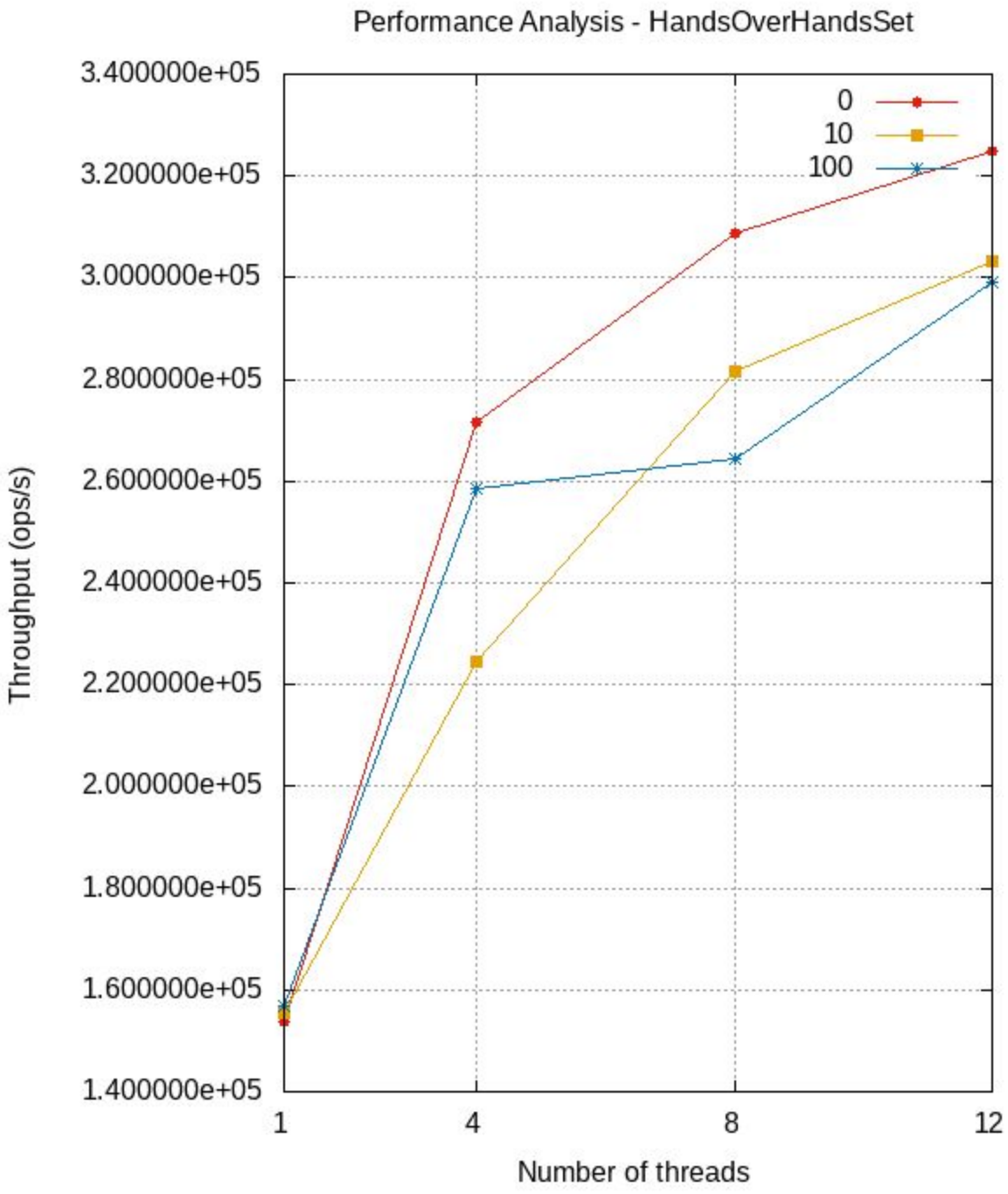
As can be seen, the HandsOverHandsSet and CoarseGrainedListBasedSet algorithms do not work very well with multithreading as they lose efficiency as the number of threads present increases, about 2.4 and 3 times respectively, compared to their throughput presented in the presence of a single thread. On the other side, the LazyLinkedListSortedSet algorithm benefits from multithreading since its efficiency increases as the number of threads present increases, and in this experiment the algorithm presents an efficiency increase of 8.7 times over the throughput presented by a single thread.

Algorithm performance analysis

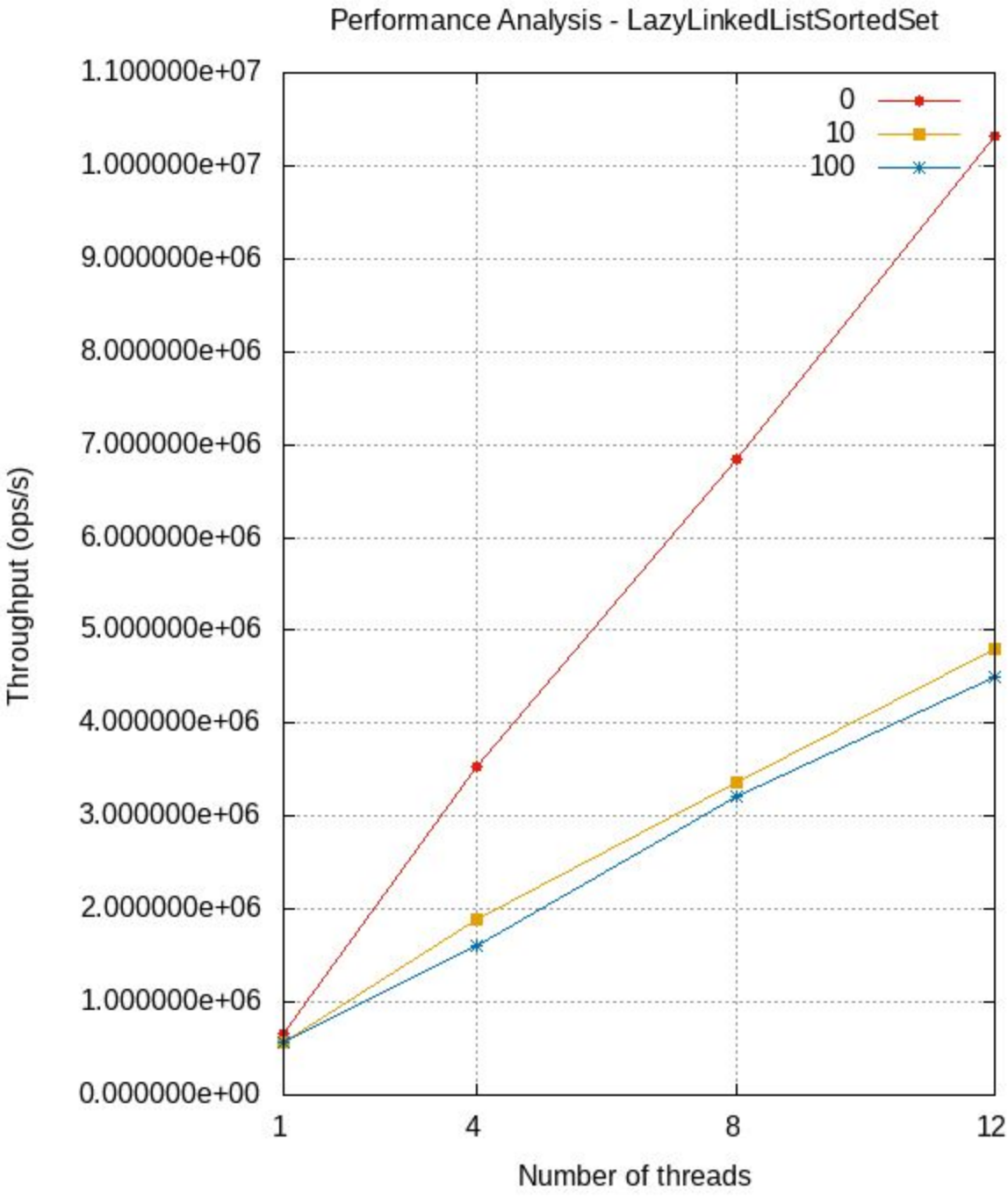
For this analysis, it will be analyzed each of the 3 algorithms separately by setting the LIST_SIZE to 1000 and varying the UPDATE_RATIO between 0, 10 and 100.



The way Locks are implemented in the CoarseGrainedListBasedSet algorithm impacts its efficiency in a very costly way, because as it can be seen, as the number of threads implemented increases the efficiency of the algorithm decreases. Therefore, the algorithm is not optimal in a high number of threads applications.



The multithreading situation is better handled by the HandsOverHandsSet algorithm, because by locking only two nodes per update, concurrency is possible when different threads work on different nodes. However, locks still present a cost for the efficiency of the algorithm.



The LazyLinkedListSortedSet algorithm benefits at best from multithreading since it is wait-free.

Computer specifications

```
Architecture :                            x86_64
Mode(s) opératoire(s) des processeurs :   32-bit, 64-bit
Boutisme :                                Little Endian
Tailles des adresses:                     39 bits physical, 48 bits virtual
Processeur(s) :                           12
Liste de processeur(s) en ligne :         0-11
Thread(s) par cœur :                      2
Cœur(s) par socket :                      6
Socket(s) :                               1
Nœud(s) NUMA :                            1
Identifiant constructeur :                GenuineIntel
Famille de processeur :                   6
Modèle :                                  158
Nom de modèle :                           Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Révision :                                10
Vitesse du processeur en MHz :            1000.322
Vitesse maximale du processeur en MHz :   4600,0000
Vitesse minimale du processeur en MHz :   800,0000
BogoMIPS :                                6384.00
Virtualisation :                          VT-x
Cache L1d :                               32K
Cache L1i :                               32K
Cache L2 :                                256K
Cache L3 :                                12288K
Nœud NUMA 0 de processeur(s) :            0-11
```

Links

- https://perso.telecom-paristech.fr/kuznetso/SLR206/class02-list-set.pdf
- https://perso.telecom-paristech.fr/kuznetso/SLR206/class01-intro.pdf
- https://www.wikiwand.com/fr/T%C3%A9l%C3%A9com_Paris