SLR206

PROJECT

Optimistic Lock-Based List-Based Set Implementations

KANG Jiale

XIE Shifeng

October 18, 2023

Contents

1 Hand-Over-Hand Algorithm

1.1 Implementation

```java
package linkedlists.lockbased;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import contention.abstractions.AbstractCompositionalIntSet;

public class HandOverHandListIntSet extends AbstractCompositionalIntSet
    {

    // sentinel nodes
    private Node head;
    private Node tail;

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

    /*
     * Insert
     *
     * @see contention.abstractions.CompositionalIntSet#addInt(int)
     */
    @Override
    public boolean addInt(int item){
    head.lock();
    Node pred=head;
    Node curr=pred.next;
    try {
      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
     *
     * @see contention.abstractions.CompositionalIntSet#removeInt(int)
     */
    @Override
    public boolean removeInt(int item){
    head.lock();
    Node pred=head;
    Node curr=pred.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;}
        return false;
      } finally{
        curr.unlock();
      } }
    finally{
```

```
      pred.unlock();
    }
    }

    /*
     * Contains
     *
     * @see contention.abstractions.CompositionalIntSet#containsInt(int)
     */
    @Override
    public boolean containsInt(int item){
    head.lock();
    Node pred=head;
    Node curr=pred.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();
    }
    }

    private class Node {

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

```
      this.lock.lock();
    }

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

    public int key;
    public Node next;
    private final Lock lock;
  }

  @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;
    }
}
```

1.2 Proof of Safety

In the case of Hand-Over-Hand Algorithm, the primary safety property is mutual exclusion: only one thread can access a critical section at a time.

This algorithm uses a fine-grained locking strategy where each node in the linked list is associated with a lock. When a thread wants to access a node in the list, it acquires the lock associated with that node. If another thread holds the lock, the first thread must wait until the lock is released. Because of this locking strategy, when one thread holds the lock for a particular node, no other thread can simultaneously hold the lock for the same node.

1.3 Proof of Liveness

Liveness of the Hand-Over-Hand Algorithm means that threads eventually make progress and do not get stuck in a waiting state.

When a thread has completed its operations on a node, it releases the lock associated with that node. This ensures that locks are not held indefinitely. When a thread releases the lock and the node, it would go to the critical section, and another thread holds that node and lock it. This shows that every threads make progress.

2 PERFORMANCE ANALYSIS

2.1 Fixed update ratio 10% and varying list size

We fixed the update ratio to 10% and changed the list size in $\{100, 1000, 10000\}$, changed number of threads in $\{1, 4, 6, 8, 10, 12\}$.

As shown in figure 1, Coarse Grained looks more stronger under the condition of less List Size and single thread. We propose an hypothesise that when using one thread, Coarse Grained Algorithm no needs to lock, while facing multiple threads, it locks the majority.

And when list size is smaller, it release the lock faster because of less operations in each list.

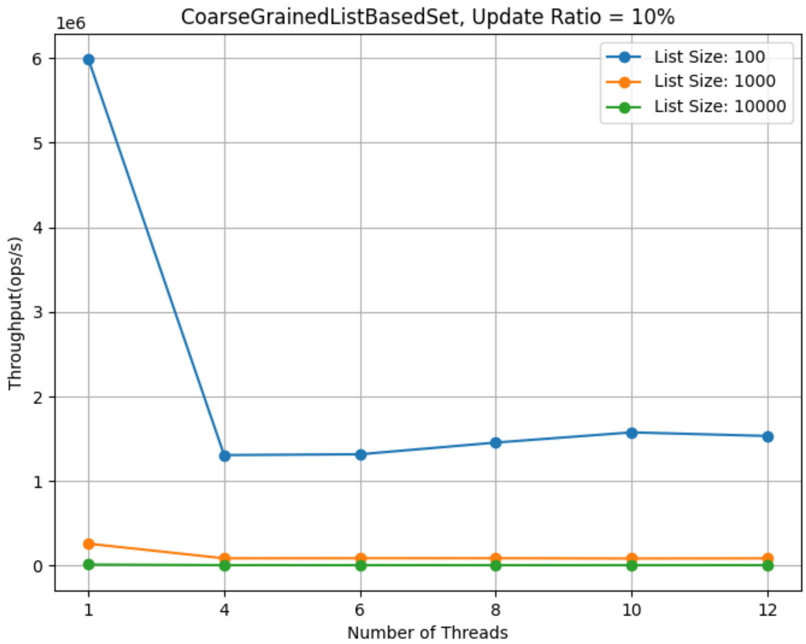

Figure 1: Coarse Grained

In the figure 2, Hand-Over-Hand Algorithm seems less affected by the changing of list size. But for the condition of single thread and less list size, it shows the best. The reason we proposed for this result is similar to the Coarse Grained Algorithm, which are using less locks and waiting less for other threads.

On the contrary, Lazy Linked Algorithm works better under the condition of multiple threads, as shown in figure 3. This is because Lazy Linked Algorithm is a non-blocking synchronization strategy. Non-blocking algorithms ensure that every process is correct.

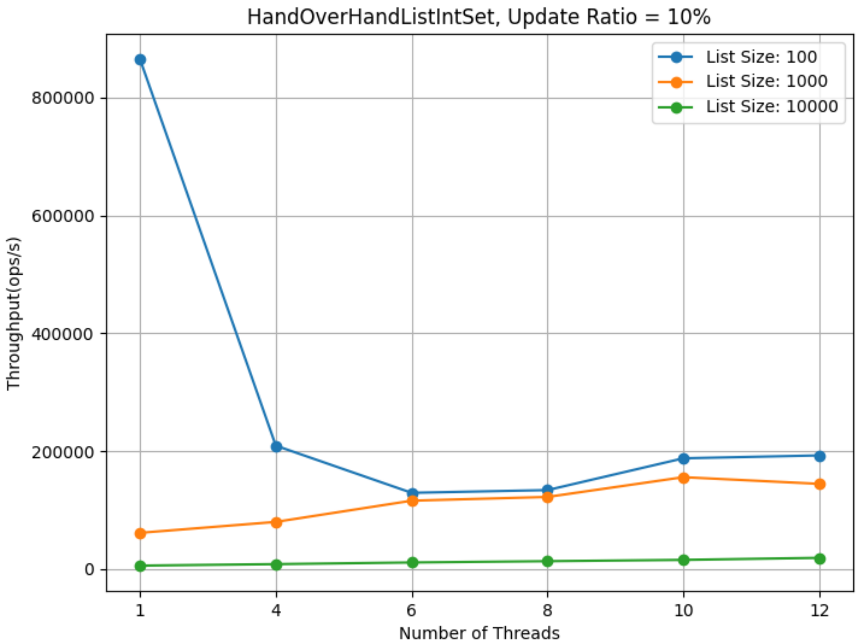

Figure 2: Hand-Over-Hand

Therefore, it is more efficient when using multiple threads.

2.2 Fixed list size 100 and varying update ratios

We fixed the list size to 100 and changed the update ratios in $\{0, 10\%, 100\%\}$, changed number of threads in $\{1, 4, 6, 8, 10, 12\}$.

For Coarse Grained Algorithm, as shown in figure 4, throughput varying less with update ratio. And single thread seems the most efficient condition. In the case of multiple threads, there is much room for improvement of the algorithm.

For Hand-Over-Hand Algorithm shown in figure 5, we proposed that throughput and update ratio is nearly independent (update ratio doesn't make big difference). The only reason decide the throughput is the number of threads.

For Lazy Linked Algorithm, update ratio affects a lot to the throughput, as shown in figure 6. Less update ratio will lead to more efficiency. And normally it works better when there are multiple threads.

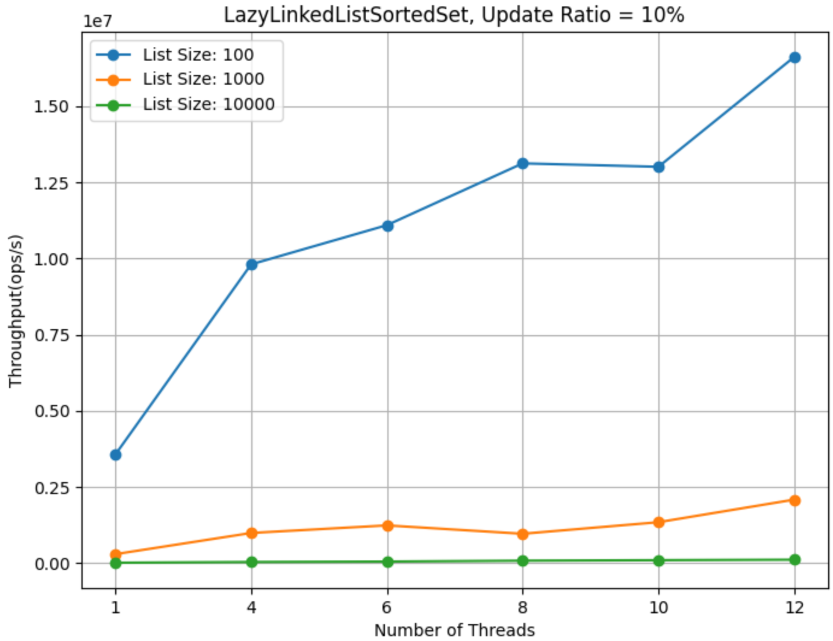

Figure 3: Lazy Linked

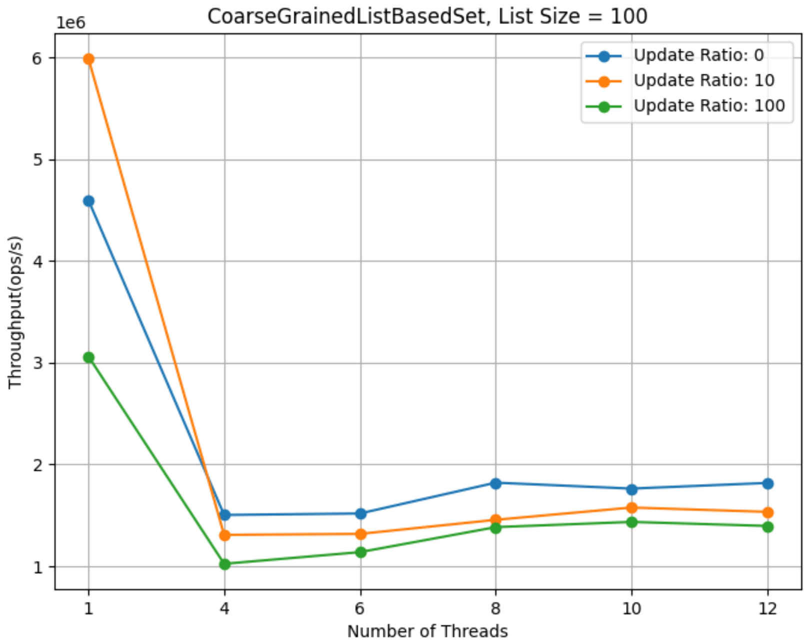

Figure 4: Coarse Grained

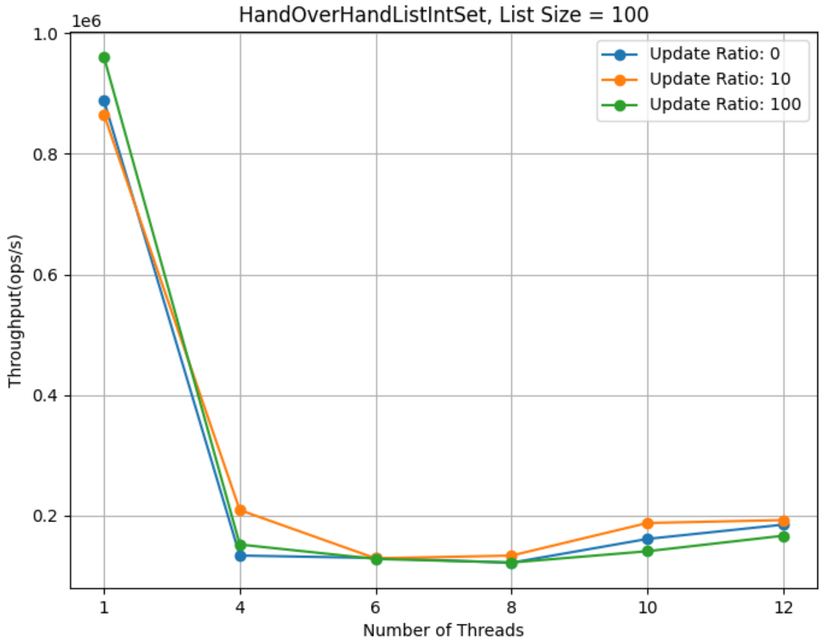

Figure 5: Hand-Over-Hand

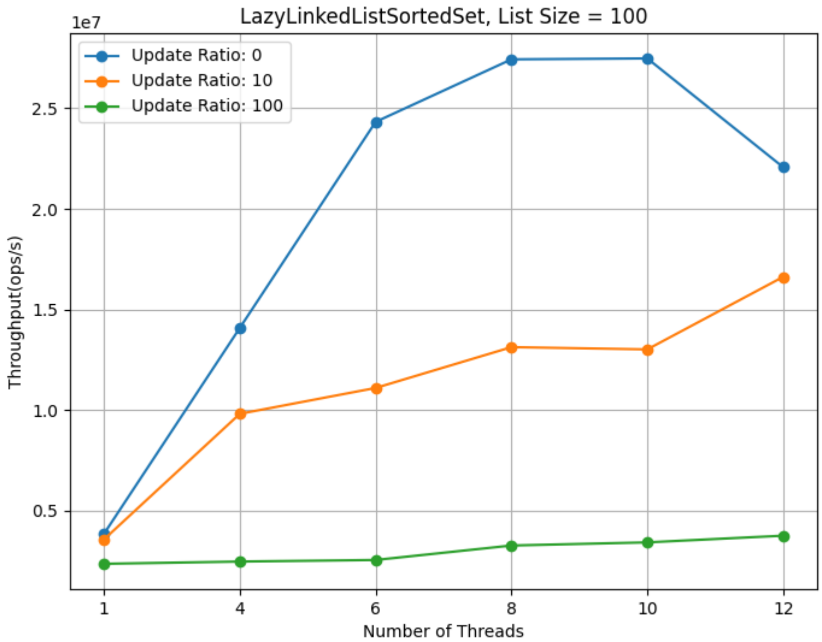

Figure 6: Lazy Linked

2.3 Fixed update ratio 10% and list size 1000

We fixed the update ratio to 10% and list size to 1000, changed number of threads in $\{1, 4, 6, 8, 10, 12\}$.

Apparently, as figure 7 described, Lazy Linked Algorithm is the best algorithm among these three and it goes better when we increase threads, proving that it gets a big benefit with concurrency operations. Hand-Over-Hand Algorithm also rises throughput with the increase of the threads but do not change as large as Lazy Linked Algorithm. While Coarse Grained Algorithm goes decrease with the change of threads number.

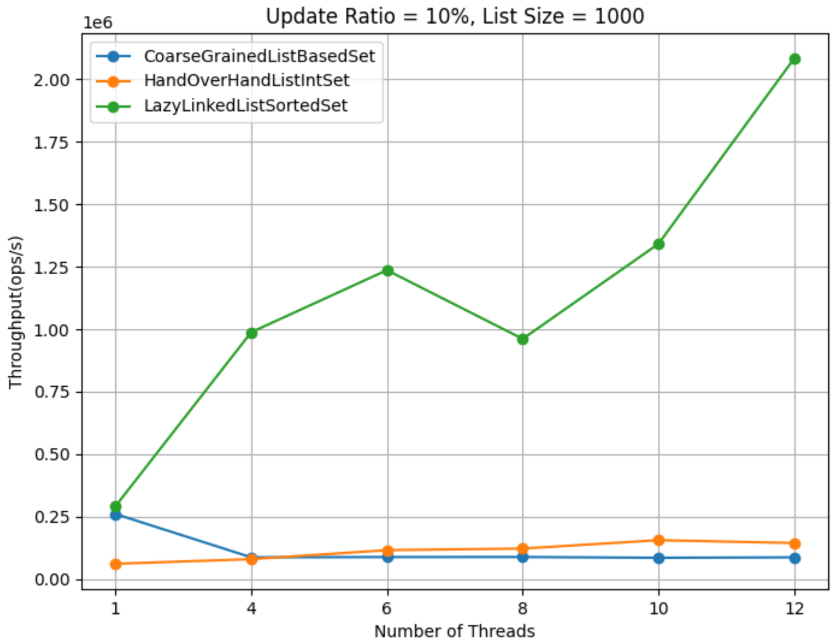

Figure 7: Three Algorithms

3 System Infomation

All the test are implemented in a remote server **@lamedell14.enst.fr**. Use command `lscpu` to check system CPU information.

```
Architecture:                x86_64
  CPU op-mode(s):            32-bit, 64-bit
  Address sizes:             43 bits physical, 48 bits virtual
  Byte Order:                Little Endian
CPU(s):                      128
  On-line CPU(s) list:       0-127
Vendor ID:                   AuthenticAMD
  Model name:                AMD EPYC 7542 32-Core Processor
    CPU family:              23
    Model:                   49
    Thread(s) per core:      2
    Core(s) per socket:      32
    Socket(s):               2
    Stepping:                0
    BogoMIPS:                5789.50
    Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep
   mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx
   mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl
   nonstop_tsc cpuid extd_apicid aperfmperf rapl pni pclmulqdq monitor
   ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand
   lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3
   dnowprefetch osvw ibs skinit wdt tce topoext perfctr_core perfctr_nb
    bpext perfctr_llc mwaitx cpb cat_l3 cdp_l3 hw_pstate ssbd mba ibrs
   ibpb stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 cqm rdt_a rdseed adx
    smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves cqm_llc
   cqm_occup_llccqm_mbm_total cqm_mbm_local clzero irperf xsaveerptr
   rdpru wbnoinvd amd_ppin arat npt lbrv svm_lock nrip_savetsc_scale
   vmcb_clean flushbyasid decodeassists pausefilter pfthreshold avic
   v_vmsave_vmload vgif v_spec_ctrl umip rdpid overflow_recov succor
   smca sme sev sev_es
Virtualization features:
  Virtualization:            AMD-V
Caches (sum of all):
  L1d:                       2 MiB (64 instances)
  L1i:                       2 MiB (64 instances)
  L2:                        32 MiB (64 instances)
  L3:                        256 MiB (16 instances)
NUMA:
```

```
  NUMA node(s):           2
  NUMA node0 CPU(s):      0-31,64-95
  NUMA node1 CPU(s):      32-63,96-127
Vulnerabilities:
  Itlb multihit:          Not affected
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Not affected
  Retbleed:               Mitigation; untrained return thunk; SMT
   enabled with ST
                          IBP protection
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled
   via prctl
                           and seccomp
  Spectre v1:             Mitigation; usercopy/swapgs barriers and
   __user pointer
                           sanitization
  Spectre v2:             Mitigation; Retpolines, IBPB conditional,
   STIBP always-
                          on, RSB filling, PBRSB-eIBRS Not affected
  Srbds:                  Not affected
  Tsx async abort:        Not affected
```