CALL - A Lock-Free Concurrent Array Linked List

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Abstract

Lock-free shared data structures implement concurrent objects without the use of mutual exclusion, thus providing more robust performance. We present a new lock-free algorithm for an unordered multiset that combines the array and list-based approaches. The algorithm takes advantage of the array's fast sequential access, and the flexibility of the linked list. All three main operations, search, insertion, and deletion, have lock-free progression, making it a non-blocking data structure. We provide an implementation of the basic algorithm, and two variants with multiple optimizations, all of which can be used to implement a *multiset* or a *set*. Our experimental results with a microbenchmark show, that the optimized version of the algorithm outperforms other similar data structures, such as the CopyOnWriteArrayList and ConcurrentLinkedQueue. The presented algorithm shows good scalability under high contention, particularly when used with a large amount of items.

Categories and Subject Descriptors Concurrent computing methodologies [*Computing methodologies*]: Concurrent algorithms

General Terms Algorithms, Design, Performance

Keywords Lock-Free, Array, Linked List

1. Introduction

Two of the main building blocks for storing data in memory are, the array and the linked list. Arrays, due to its inherent structure, provide fast searches, because memory allocation and access on modern computer architectures favors data structures that rely on locality [13, 14]. For access patterns with a high number of mutative operations, the only solution until now was to create a new array and copy the data from the previous one, the Copy-On-Write pattern (COW) [9, 10].
On the other hand, linked lists can adapt dynamically, but given that data is not stored sequentially, they suffer from slow traversals. This will impact the performance of lookups, and even insertions and deletions when done in the middle of the list, depending on how sparsely the data is stored in memory.

Several lock-free concurent linked list algorithms are known [2–4, 11, 12, 15–17]. One of the most commonly deployed concurrent data structures based on a linked list is the one by Michael and Scott [12]. It is a simple and fast lock-free queue. Both insertion and deletion is only performed at the end or start of the list, respectively. Looking specifically to the `add()` operation, the insertion is performed at the tail of the list, where the *tail* is the last node on the list. The operation will be finished when the item is appended to the list, and can occasionally help other `add()` operations to complete, by updating the tail to reference the current last node on the list.

Another concurrent linked-list, is Harris's linked list [3, 4]. It is a lock-free ordered linked-list, thus allowing insertion and deletion in the middle of the list. Deletion is performed in two stages, the first action is to mark the node containing the item to be deleted, by changing the `next` field of that node. This action will *logically remove* the item, and finally, the node is removed by changing the previous node to reference the next node. The entire mechanism guarantees the correct removal of an item at the same time as other insertions or deletions occur.

In this document, we present a concurrent *unordered multiset* with lock-free operations, which we named ConcurrentArrayLL (CALL). Sometimes called a *bag*, the multiset has several known concurrent implementations, such as the CopyOnWriteArrayList [9] and ConcurrentLinkedQueue [8].

A general description of the data structure is shown in section 2. In section 3, we present the basic algorithm in detail (CALLBase), followed by a discussion of correctness and progress conditions on section 4. The two optimized algorithms (CALLItem and CALL) are then shown in section 5, followed by the *set* implementation on section 6. In section 7 we present our empirical evaluation, and conclude in section 8.

2. General Description

A *list* represents a collection of items, where each item has a position in the list. One possible implementation of a list, is a singly linked list that is formed by a collection of nodes, each containing an item of the list, where each node references the following position of the list through a field `next`. A collection of items can be classified as a *multiset* or *set*, in case duplicate items are allowed or not, respectively. We present our concurrent array linked-list algorithm in the context of an unordered multiset collection. This algorithm maintains the order of insertion, where an insertion occurs always at the end of the list. The `CALL` data structure does not impose any constraint on the data, data does not have to be able to be ordered or classified based in any intrinsic characteristic. It is suitable for applications whose data have many searchable characteristics or data, like images or text, that have no specific characteristic that can be exploited to impose a structure on the data.

For our purposes, a *multiset* provides three methods:

– The `add(x)` method adds item x to the multiset, returning always true even if an existing indistinguishable item is already present in the multiset.

– The `remove(x)` method removes item x from the multiset, returning true if and only if x was in the multiset.

– The `contains(x)` method returns true if and only if the multiset contains item x.

For each method, we say that a call is successful if it returns true, and unsuccessful otherwise.

`CALL` takes advantage of two data structures, array and linked list, using them where they are most suitable. An array is a static data structure that presents several constraints every time a modification is performed. For instance, the removal of an item on an array, requires that all items positioned after the index of the removed item, be shifted to the left. This operation can not be performed at the same time a search is being executed, as it could result in an incorrect behavior, not finding an item that is present in the data structure. An insertion can only be executed concurrently, if it is done at the end of the array, and its length is sufficient, otherwise a new array has to be created and the already refered COW pattern can be used to allow lock-free progress. In such approach, each thread attempting an insertion creates an array instance, but only one thread will be successful, making it impractical and instead, a mutual exclusion lock is prefered [9], allowing only one mutative operation at a time.

Having this into consideration, we have developed a solution that avoids these limitations. The solution is to have a composed structure, `ArrayLL`, that has an array and a linked list, as shown in Figure 1. All `add()` operations are executed on the linked list at the tail of the list, using the algorithm presented in [12]. The `remove()` operation is achieved through a Compare-And-Swap (CAS) instruction on a variable, `state`, that every node of the linked list has. The actual update of the structure is decoupled from the write operations. The structure is put up to date using a method called `rebuildAttempt()`. The `rebuildAttempt()` method can be called by any of the operations, `add()`, `remove()` or `contains()`, and its job is to copy all the nodes to a new array, in a way similar to the COW pattern. The thread assigned the rebuild task is decided through an election mechanism.

This approach, when compared with the COW technique, results in large performance gains when the workload of write/read operations increases. Its main advantage comes from the aggregation of multiple write operations before rebuilding the structure. On top of that, all operations have lock-free [6] progress condition.

2.1 Election Mechanism

Through an election consensus, using a simple compare-and-swap `CAS(NOT_REBUILD, REBUILD)`, one of the threads is elected to go through a different code path. This alternate code path will correspond to the execution of the `rebuildAttempt()` method in our algorithm. Once the `rebuildAttempt()` finishes, the elected thread will set the consensus variable back to `NOT_REBUILD`. Such a mechanism guarantees that a method runs in isolation from other threads. All the other threads that didn't win the election will proceed by executing their operation on the data structure. In summary, the elected thread is selected to do an extra work, which consists of putting the data structure up to date, on top of executing its own operation.

The drawback of this mechanism is that the execution of that extra work is dependent on the elected thread to finish. In case of a failure of the elected thread, from that point on, no other thread will be able to be elected and execute that extra work. Fortunately, in the case of our algorithm, it is possible to adapt the election mechanism to allow the election of a subsequent thread after a `MAX_ATTEMPTS` number of iterations, allowing several threads to execute the extra work at the same moment in time. The modified election mechanism is composed of rounds, where each round starts when there is an initial thread that changes through a CAS instruction the consensus variable from `NOT_REBUILD` to `ThreadID`, where the `ThreadID` is an unique identifier for every thread, like for example the value returned by Java's `Thread.currentThread().getId()`. During this same round, other threads can be elected by changing through a CAS instruction to their own id, but only after a sufficient amount of iterations, `MAX_ATTEMPTS`, have elapsed since the last election. The round will be closed once the consensus variable is set back to `NOT_REBUILD`.

```
class CALL<E> {
    static final int MAX_ATTEMPTS;
    volatile int rebuildGuard;
    volatile int numRebuildAttempts;
    volatile ArrayLL<E> validArrayLL;
    volatile Node<E> tail;
}
class ArrayLL<E> {
    Node<E>[] arrayNode;
    int inUseLength;
    Node<E> head;
}
class Node<E> {
    final E item;
    volatile Node<E> next;
    volatile int state;
}
```

Figure 1. The CALL data structure is composed of two classes which we show in Java code: `ArrayLL` and `Node`.

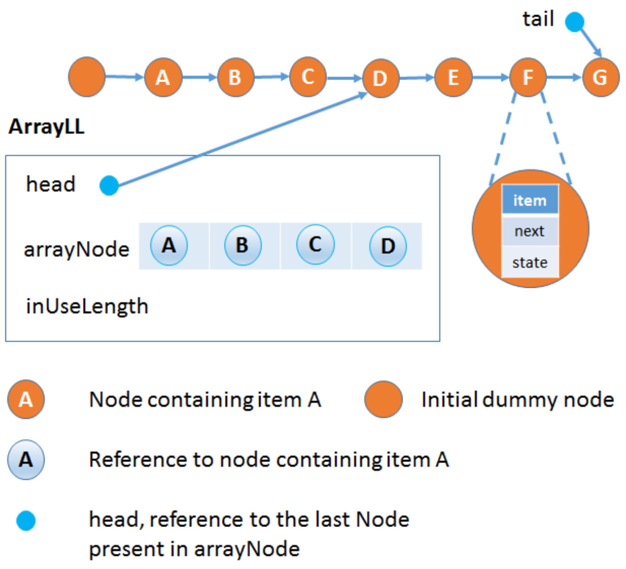

Figure 2. Building blocks of CALL.

3. Algorithm Design

The `CALL` data structure is represented in Figure 1. This data structure has a variable `validArrayLL`, that is an instance of the `ArrayLL` structure, a variable `tail`, and two variables that are used by `rebuildAttempt()`, `rebuildGuard` and `numRebuildAttempts`, to manage the threads that will be allowed to update `validArrayLL`. The `tail` variable is used by the `add()` operation to insert items at the end of the list.

One of the main building blocks of the `CALL` is a structure named `ArrayLL`. The `ArrayLL`, shown in Figure 1, is composed of three members: an array of nodes, a field named `inUseLength`, and a reference to a node in the linked list, `head`. The array, `arrayNode`, stores references to the nodes of the linked list. The field named `inUseLength` corresponds to the number of positions that are currently in use on the array, and it is always lower than, or equal to the size of the array. The third member is `head`, a reference to the last node that was copied to the `arrayNode`. The `head` can be seen as a *logical* linked list, where insertion is only allowed on the `tail` node. All the nodes of the linked list are composed of three members: the reference to the item being stored, the reference to

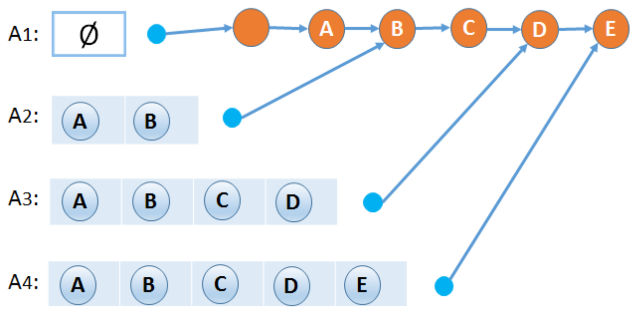

Figure 3. Several `ArrayLL` instances, A_1 to A_4, created by the `rebuildAttempt()` method. It is possible for all A_1 to A_4 instances to be active, being accessed by a currently running operation. In this figure, A_4 corresponds to `validArrayLL`, the instance that will be accessed by new operations.

the next node, and the state of the node, which can be `INUSE` or `REMOVED`. Figure 2 shows how all these components fit together.

The `validArrayLL` allows concurrent access to all three operations, `add()`, `remove()` and `contains()`. This is possible because there is no real modification of the structure itself. Specifically, all `add()` operations occur at the tail of the linked list, following the algorithm presented on [12], and the `remove()` operation marks the item as removed by setting the node's `state` to `REMOVED`. The real modification of the structure is done by an auxiliary method named `rebuildAttempt()`, and the thread that is responsible for this task is chosen throught an election mechanism. The election occurs when there is one thread that successfully changes the `rebuildGuard` variable throught a CAS instruction. Other threads can be elected in subsequent rounds to update the data structure, but only after `numRebuildAttempts` has reached `MAX_ATTEMPTS`. After winning the election, the thread can proceed to create a new instance of `ArrayLL` that will not contain items marked as `REMOVED`, and the references to the nodes of the logical linked list, starting at `head`, will be copied to the new `arrayNode`.

The `rebuildAttempt()` procedure consists of copying all the references of the nodes from `validArrayLL` to a new instance of `ArrayLL` and follows a best-effort approach. The size of the new array will be estimated from `validArrayLL` by adding its `inUseLength` and measuring the size of the logical linked list. This estimated size may be smaller than what would be necessary to store references to all the nodes in the list, because at the same time, `add()` operations are being executed at the `tail` of the linked list. This will result in a new array that is closer to the size of the list than before. Only items that are being added or removed during a rebuild, may be left to be copied on the next call to `rebuildAttempt()`. This procedure follows the COW mechanism, where there can be several active instances of `ArrayLL` still being referenced by ongoing threads. This solution doesn't manage the release of memory, being dependent of a system that provides a garbage collector.

Regarding the linked-list, it is important to notice that there is only one instance, which is shared by all instances of `ArrayLL`, but depending on the `head` of each `ArrayLL`, several *logical* linked lists may co-exist, where each logical linked list starts at the node referenced by `head`, as can be seen in Figure 3. Notice that all four instances A_1 to A_4, shown in this figure, are *equivalent*, in the sense that they represent the same set of items.

The `tail` is always reachable by any of the references `head`, which guarantees that all items present on the logical linked list and items that are being inserted by any `add()`, are made visible to all `remove()` and `contains()` operations executing at any instance of `ArrayLL`.

3.1 Basic Algorithm

In this section we describe the basic algorithm. This data structure provides four major methods. In addition to the methods `add(`$item$`)`, `remove(`$item$`)`, and `contains(`$item$`)` defined in section 2, there is also a `rebuildAttempt()` method. The `rebuildAttempt()` method makes a new instance of `ArrayLL` without the nodes that have been previously marked as `REMOVED`, and adds to the array the nodes present on the logical linked list, starting at `head`.

3.1.1 add(x)

The `add(x)` method is similar to the `enqueue(x)` method described in [12], and it is shown in Algorithm 1. Notice that the `Node` is initially allocated with `state` at `INUSE`.

Algorithm 1 Algorithm for `add()` operation

```
1: function ADD(item)
2:     newNode = new Node(item, INUSE);
3:     while true do
4:         localTail = tail.get();
5:         node = localTail.next.get();
6:         if localTail == tail.get() then
7:             if node == null then
8:                 if localTail.next.CAS(null, newNode) then
9:                     tail.CAS(localTail, newNode);
10:                    return true;
11:                end if
12:            else
13:                tail.CAS(localTail, node);
14:            end if
15:        end if
16:    end while
17: end function
```

3.1.2 contains(x)

As shown in Algorithm 2, the `contains(`x`)` method must first read the current value of the `validArrayLL` variable and from then on, always use the instance of `ArrayLL` thus obtained. It will search for the item x first on the array of nodes, `arrayNode`, and if it doesn't find it there, or if it finds x but it is logically removed (with `state` at `REMOVED`), then it will continue searching for x on the linked list, starting at `head` and stopping only when it finds a node whose `next` is `null`. The return value will be *true* if a node containing the item x is found and it is in state `INUSE`, or *false* otherwise.

Another task of the `contains()` method, is to call `rebuildAttempt()`, as shown on line 4 of Algorithm 2. In this implementation, the `rebuildAttempt()` method is only called when a new item was inserted after the last rebuild, this happens because there is no way to know if a `remove()` operation has occurred, without traversing the list searching for nodes with state `REMOVED`. In case the removal of an item has occurred, this solution doesn't guarantee a timely physical elimination of nodes containing items already marked as `REMOVED`, which can impact performance upon traversing the list. This event is avoided by calling the `rebuildAttempt()` method also from `remove()`, as shown in Algorithm 3.

3.1.3 remove(x)

Much like `contains(`x`)`, the `remove(`x`)` method scans the nodes in the `arrayNode`, and then on the linked list, searching for item x. Once it finds the node that references x, it will return *true* if it succesfully changes the `state` of the node from `INUSE` to `REMOVED` using a CAS, as shown in lines 15 and 31 of Algorithm 3. If it reaches the end of the linked list without finding a node with the x item, or without succeeding in the CAS operation, it will return *false*. One benefit of logically marking the node as `REMOVED`, is that we can separate this step from the disruptive physical changes to the structure, such as disconnecting the node, or removing it from

Algorithm 2 Algorithm for `contains()` operation

```
function CONTAINS(item)
    aLL = validArrayLL.get();
    if aLL.head.get() $\neq$ tail.get() then
        REBUILD( );
    end if
    return FINDINARRAY(aLL, item) || FINDINLL(aLL, item);
end function

function FINDINARRAY(aLL, item)
    arrayNode = aLL.arrayNode;
    for i = 0; i $<$ aLL.inUseLength; i++ do
        if arrayNode[i].item == item then
            if arrayNode[i].state.get() $\neq$ REMOVED then
                return true;
            end if
        end if
    end for
    return false;
end function

function FINDINLL(aLL, item)
    node = aLL.head;
    while true do
        if (node = node.next.get()) == null then
            return false;
        end if
        if node.item == item then
            if node.state.get() $\neq$ REMOVED then
                return true;
            end if
        end if
    end while
end function
```

Algorithm 3 Algorithm for `remove()`

```
function REMOVE(item)
    aLL = validArrayLL.get();
    isRm = (RMFROMARRAY(aLL, item) || RMFROMLL(aLL, item));
    if isRm then
        REBUILD( );
    end if
    return isRm;
end function

function RMFROMARRAY(aLL, item)
    arrayNode = arrayLL.arrayNode;
    for i = 0; i $<$ aLL.inUseLength; i++ do
        node = aLL.arrayNode[i];
        if node.item == item then
            if node.state.get() $\neq$ REMOVED then
                if node.state.CAS(INUSE, REMOVED) then
                    return true;
                end if
            end if
        end if
    end for
    return false;
end function

function RMFROMLL(aLL, item)
    node = aLL.head;
    while true do
        if (node = node.next.get()) == null then
            return false;
        end if
        if node.item == item then
            if node.state.get() $\neq$ REMOVED then
                if node.state.CAS(INUSE, REMOVED) then
                    return true;
                end if
            end if
        end if
    end while
end function
```

the `arrayNode`. As already mentioned, the `rebuildAttempt()` is also called from de `remove()` method, and will be attempted only upon a successful removal.

3.1.4 rebuildAttempt()

`rebuildAttempt()` is the method responsible for creating a new instance of `ArrayLL`, and copying the contents from the `validArrayLL` instance into it. Several threads may be executing the rebuild operation at a given time, but the procedure still follows the election mechanism. For every round, after the first thread is elected, all subsequent threads can only attempt to be elected when the global counter `numRebuildAttempts` is over `MAX_ATTEMPTS`. In that case, a new election will take place where only one of those threads will win the election, restarting the global counter for the next round. This procedure prevents all threads from constantly updating the data structure and, at the same time, guarantees that if an elected thread is for some reason unable to finish the `rebuildAttempt()` method, after a maximum number of iterations another thread is allowed to perform the rebuild, thus conferring to the algorithm the property of being fault tolerant.

Having in mind that several threads can be in charge of creating a new instance of ArrayLL that will replace the current `validArrayLL`, it will be the last modification of `validArrayLL` that will remain visible, but it is important to notice that all `ArrayLL` instances created by all elected threads are valid MultiSets, all of them being equivalent in the sense that they represent the same MultiSet. The election mechanism is not essential for the correctness of the algorithm, it would be valid to allow all threads to execute `rebuildAttempt()`, all competing to update the data structure, but this would cause an enormous waste in memory allocation and the garbage collection associated with it, and would also impact performance.

When copying nodes from the array of the old instance to the new one, the rebuild operation will *unlink* — physically remove the node — any nodes marked as `REMOVED` from the linked list. This is done by setting the `next` field of the previously `INUSE` node to reference the node after the `REMOVED` node.

Notice that if the first node in the array is `REMOVED`, it will not be unlinked, because there is no reference to the previous node to be able to unlink it. This will not result in memory leakage, because once the older `ArrayLL` instances are no longer being used, there will be no reference to that particular node, nor to any of the nodes that link to it, and the automatic GC will be able to delete it.

After copying the contents of the array of nodes, it is still necessary to copy the nodes added to the linked list. We decided to limit the copy up to the tail of the list, so as to guarantee that the tail always references a node in the linked list, which will be useful to the implementation of `addIfAbsent()` shown in section 6.

4. Correctness and Progress Conditions

4.1 Progress Conditions

We show that a given method is lock-free [1] if it guarantees that at least one among all concurrent methods finishes in a finite number of steps. We show that a given method is wait-free population oblivious [6] by showing that every call to that method finishes its execution in a finite number of steps, independently of the number of threads.

An `add()` operation loops only if one of the conditions fails in lines 6, 7, or 8 of Algorithm 1. Any of those conditions will fail if and only if another thread has succeeded in adding a new node to the list, thus making `add()` lock-free.

The `contains()` operation is lock-free because it has no loops, and consists of three method calls to: `findInArray()`, `findInLL()` and `rebuildAttempt()`, all of which are at worst lock-free. The `findInArray()` method is wait-free [5] because

Algorithm 4 Algorithm for `rebuildAttempt()`

```
 1: function REBUILD
 2:     currentOwner = rebuildGuard.get();
 3:     if (currentOwner == NOT_REBUILD ||
 4:         numRebuildAttempts ¿= MAX_ATTEMPTS) then
 5:         numRebuildAttempts = 0;
 6:         if !rebuildGuard.CAS(currentOwner, ThreadID) then
 7:             numRebuildAttempts++;
 8:             return false;
 9:         end if
10:     else
11:         numRebuildAttempts++;
12:         return false;
13:     end if
14:     origaLL = validArrayLL.get();
15:     newSize = origaLL.inUseLength + sizeOfLL(origaLL.head);
16:     newaLL = new ArrayLL(new Node[newSize],origaLL.head,0);
17:     COPYNODE2ARRAYLL(origaLL, newaLL);
18:     if origaLL.head ≠ tail.get() then
19:         COPYLL2ARRAY(newaLL);
20:     end if
21:     validArrayLL.set(newaLL);
22:     rebuildGuard.compareAndSet(ThreadID, NOT_REBUILD);
23: end function

24: function COPYNODE2ARRAYLL(srcArrayLL, dstArrayLL)
25:     pos = 0;
26:     srcArrayNode = srcArrayLL.arrayNode;
27:     dstArrayNode = dstArrayLL.arrayNode;
28:     for i = 0; i < srcArrayLL.inUseLength; i++ do
29:         node = srcArrayNode[i];
30:         if node.state.get() ≠ REMOVED then
31:             dstArrayNode[pos] = node;
32:             pos++;
33:         else
34:             if pos ¿ 0 && i ≠ srcArrayLL.inUseLength-1 then
35:                 dstArrayNode[pos-1].next.set(srcArrayNode[i+1]);
36:             end if
37:         end if
38:     end for
39:     dstArrayLL.inUseLength = pos;
40: end function

41: function COPYLL2ARRAY(newaLL)
42:     pos = newaLL.inUseLength;
43:     node = newaLL.head;
44:     dstArrayNode = newArrayLL.arrayNode;
45:     if node.state == REMOVED && pos ¿ 0 then
46:         newaLL.arrayNode[pos-1].next.set(node.next.get());
47:     end if
48:     while pos < newaLL.arrayNode.length do
49:         node = node.next.get();
50:         localTail = tail.get();
51:         if node.state ≠ REMOVED then
52:             dstArrayNode[pos] = node;
53:             pos++;
54:         else if node ≠ localTail && pos ¿ 0 then
55:             dstArrayNode[pos-1].next.set(node.next.get());
56:         end if
57:         if node == localTail then
58:             break;
59:         end if
60:     end while
61:     newaLL.inUseLength = pos;
62:     newaLL.head = node;
63: end function
```

it will always loop for a finite number of steps, which depend on `inUseLength`. The `findInLL()` method is lock-free because it loops until it finds a valid node with the `item` that is being searched for, in lines 25 and 26 of Algorithm 2, or until it finds a node with `next` equal to `null` in line 22 of Algorithm 2. This will occur in a finite number of steps, unless other threads are successfully adding new nodes to the last node of the linked list, which means some other concurrent method is making progress.

The `rebuildAttempt()` operation is wait-free because it has no loops, and it consists of two method calls, `copyNode2ArrayLL()` and `copyLL2Array()`, both being wait-free. The `copyNode2ArrayLL()` method is wait-free because it iterates through the array of nodes with a finite size of `inUseLength`, copying them to the new array. The `copyLL2Array()` method traverses the linked list and copies each node to the newly created array (if they are not in state `REMOVED`), until it fills up the new array, on line 48 of Algorithm 4, or until it reaches the node that was read as being the tail, on line 57 of Algorithm 4. This procedure will take a finite number of steps, thus implying that `copyLL2Array()` is wait-free.

The `remove()` operation is lock-free because it has no loops and consists of three method calls: `rmFromArray()`, `rmFromLL()`, and `rebuildAttempt()`, all of which are at worst lock-free. The `rmFromArray()` method is wait-free because it iterates through the array of nodes, which has a finite length of `inUseLength`. The `rmFromLL()` method is lock-free because it traverses the linked list until it finds a node with a matching `item`, or until it finds a node with `next` equal to `null`. This will occur in a finite number of steps, unless another thread is continuously adding new nodes to the tail of the linked list, which means the other thread is progressing, and thus implying `rmFromLL()` is lock-free.

4.2 Linearizability

Linearizability [7] is a standard correctness condition for concurrent data structures.

The linearization point for `add()` occurs in line 8 of Algorithm 1 when the CAS instruction is successful, thus changing the `next` field of the current last node in the list from `null` to reference the newly created node.

When a `remove()` operation succeeds, its linearization point is the successful CAS instruction that changes the `state` from `INUSE` to `REMOVED`, thus making the removal *visible* to other threads. If the node is in the array, it will be in line 15 of Algorithm 3 otherwise, the linearization point will occur in the linked list, in line 31 of Algorithm 3. For an unsuccessful `remove()` call, the linearization point is the point at which it reaches the end of the list, where `next` equals `null`, in line 26 of Algorithm 3.

The linearization point for a successful `contains()` occurs in either line 12 or 26 of Algorithm 2 when a matching `item` is found on a node whose `state` is `INUSE`. For an unsuccessful `contains()` call, the linearization point is the point at which it reaches the end of the list, where `next` equals `null`, in line 22 of Algorithm 2.

For the `rebuildAttempt()` operation, the linearization point occurs when the newly created `ArrayLL` instance becomes visible to other threads. This happens in the atomic set of `validArrayLL`, in line 21 of Algorithm 4.

4.3 Correctness

For brevity, on this section, we provide only an outline of the correctness proof. The correctness of the algorithm can be established by the *representation invariant*. The multiset algorithm described in this paper requires the following invariants:

- **INV1** The `tail` always references a node in the linked list.
- **INV2** The `next` member of a node will never reference `null` unless it is the `last` node.
- **INV3** The `tail` is always reachable from `head`.
- **INV4** Nodes are only inserted after the `last` node in the linked list.
- **INV5** A node is removed by changing its `state` from `INUSE` to `REMOVED`.
- **INV6** The unlink of a `REMOVED` node leaves an equivalent multiset.

4.4 Sketch of Proof

CALL starts with an empty array and a linked list containing a dummy node, that both tail and head reference to. Initially, tail is reachable from head because it is the same node. Furthermore, tail is only modified by the add() operation, that consists of changing the next field of tail to reference the new node and afterwards update tail to reference it, according to INV4. By induction, assuming tail references a node in the linked list before the add() operation executes, then the new node will also be on the linked list, and we can conclude that the updated tail will continue to reference a node in the linked list. The same argument can be used to prove that tail continues to be reachable from head after an add() operation.

In the case of a remove() operation, there is no change on the next field, according to INV5, which guarantees that tail continues to be reachable from head, and that tail references a node in the linked list, even if that node is marked as REMOVED.

After an add() operation has changed the next field from null to the newly created node, next can only be changed by the rebuildAttempt() method, to unlink nodes marked as REMOVED (this will allow a more efficient garbage collection). The unlink done in lines 35, 46 and 55 of Algorithm 4, can only occur when the node marked REMOVED has its next field that is, or was, referencing a node in the linked list. This is guaranteed by: conditions in lines 54 and 57 of Algorithm 4 that only allow nodes that precede the tail to be unlinked, and all nodes preceding tail have a non-null next field, thereby referencing a node in the linked-list; condition in line 34 guarantees the same for the unlink on the arrayNode; regarding the unlink on line 46, its purpose is to handle the special case of when head is marked REMOVED, in this case, head can only be unlinked if is different from tail, guaranteed by the condition in line 18 of Algorithm 4.

Since only nodes preceding tail are unlinked, it guarantees that tail will continue to reference a node in the linked list, even after an unlink has taken place. Also, the unlink procedure is done only on nodes preceding tail, tail not included, and the add() operation is done at the node referenced by tail, this guarantees that both tasks don't have any impact on each other executions. This concludes the proof for INV1.

Invariant 2 holds because, once the node is no longer the last node of the linked list, it means it will be referencing a new node, by INV4. As previously shown, the unlink task can only change next to reference a node in the linked list, which proofs that it will never be null again. This concludes the proof for INV2.

For every instance of ArrayLL, the head will reference the last node copied to arrayNode and will never again be changed, and at that moment, head references a node in the linked list. Nevertheless, it is possible for the node referenced by head to be unlinked from the linked list, in lines 35 or 46 of Algorithm 4. As mentioned before, the unlink of a node will only occur if the node's next field is, or was, referencing a node in the linked list, and only nodes preceding the tail can be unlinked.
By induction, if tail is reachable from head before the unlink of a node, then after the unlink of such node, even in the case the node is responsible for tail to be reached, that node will continue to reference a node in the linked list, according to INV2. We can conclude that tail is reachable from head, having in mind that the unlinked node can never be the node referenced by tail. This concludes the proof for INV3.

INV4 is guaranteed by the condition in line 8 of Algorithm 1, and no other task besides add() can take place at the tail node. INV5 is guaranteed by conditions in lines 15 and 31 of Algorithm 3.

Finally, invariant 6 holds because as previously shown, an unlink only occurs when the node marked REMOVED has a next field that references a node in the linked list, that we name $nodeX$. To conclude the proof it is necessary that $nodeX$ is a node positioned after the removed node, which is true as can be seen in lines 35, 46 and 55 of Algorithm 4.

5. Optimizations

Several improvements can be applied to the original algorithm — which we named CALLBase — to significantly increase the overall throughput.

5.1 Array of items in ArrayLL

One way to improve performance, is to add in ArrayLL an array of references to the items, arrayItem, and this way, save a pointer indirection when searching for an item, by traversing the arrayItem instead of the arrayNode. When the item that is being searched for is found, the same index must be checked on arrayNode to see if the corresponding node's state is in INUSE, and if it is not, the search must continue. This optimization will be used in the algorithms CALLItem and CALL.

5.2 Add an arrayLLState

In order to efficiently communicate the need to rebuild the current ArrayLL to the next contains() operation, we constructed a mechanism that has two references to this structure. One that is always pointing to the most recent ArrayLL instance, validArrayLL, and another named arrayLLState that can take $2t + 2$ states, where t is the number of threads: null, an empty instance of ArrayLL per thread (that is named rebuildInProgress), and valid rebuilt instance of ArrayLL per thread (that we named newaLL). It will be null if there was a removed item since the last rebuild of the data structure; rebuildInProgress when a rebuild() is currently executing. Once arrayLLState is equal to validArrayLL it means there were no remove() operations since the last rebuild.

This optimization is used in the algorithm CALL.

5.2.1 contains() can go through a fast path

When using the optimization with arrayLLState, it becomes possible for the contains() operation to guarantee that the items in arrayItem are all present in the multiset at the moment arrayLLState was read. In addition, if the linked list is empty, the arrayItem of that ArrayLL instance is a *snapshot* of the data structure, which means that the contains() needs only to search in this array, and doesn't need to search on the linked list or the array of nodes. We can conclude that the linearization point takes place when the variable arrayLLState is read. When this code path is executed, and only for this code path, the search will be wait-free population oblivious, because it is bounded by the number of entries in the array.

This code path is similar to the one used by CopyOnWriteArrayList [9], doing no acquire barriers during the search, except for a single one when the arrayLLState variable is read, and so, when there are none or very few write operations (add() or remove()), the throughput approaches that of CopyOnWriteArrayList.contains().

5.3 Optimized Algorithm

On Algorithm 5 we show the contains() method with the optimizations described above. The function findInArray() has also been modified to search on the array of items instead of the array of nodes.

Compared to the basic algorithm described in Algorithm 4, there are two extra and two changed statements in the optimized rebuildAttempt() shown in Algorithm 6, namely in lines 16, 18, 19, and 25. Before constructing an updated copy of validArrayLL

Algorithm 5 Algorithm for optimized `contains()`

```
function CONTAINS(item)
    aLL = validArrayLL.get();
    if aLL.head.next == null && aLL == arrayLLState.get()
    then
        for i=0; i < aLL.length; i++ do
            if aLL.arrayItem[i] == item then
                return true;
            end if
        end for
        return false;
    else
        REBUILDATTEMPT( );
        return findInArray(aLL, item) —— findInLL(aLL, item);
    end if
end function
```

(`newaLL`), `arrayLLState` is set to `rebuildInProgress`, thus *signalling* to other threads that a rebuild operation is currently ongoing. At the end of the rebuild, the newly created `ArrayLL` instance will become the new value of `arrayLLState` if there wasn't any change of state done by a `remove()` operation, or other threads trying to do a rebuild themselves. The rest of the algorithm is similar to the `rebuildAttempt()` method presented in section 3.1.4, except there is an extra array, `arrayItem`, to be updated.

Algorithm 6 Algorithm for optimized `rebuildAttempt()`

```
function REBUILD
    currentOwner = rebuildGuard.get();
    if (currentOwner == NOT_REBUILD ||
        numRebuildAttempts ¿= MAX_ATTEMPTS) then
        numRebuildAttempts = 0;
        if !rebuildGuard.CAS(currentOwner, ThreadID) then
            numRebuildAttempts++;
            return false;
        end if
    else
        numRebuildAttempts++;
        return false;
    end if
    origaLL = validArrayLL.get();
    newSize = origaLL.inUseLength + sizeOfLL(origaLL.head);
    newaLL = new ArrayLL((E[]) new Object[newsize],
new Node[newSize],origaLL.head,0);
    rebuildInProgress = new ArrayLL¡E¿(null, null, null, 0);
    arrayLLState.set(rebuildInProgress);
    COPYNODE2ARRAYLL(origaLL, newaLL);
    if origaLL.head ≠ tail.get() then
        COPYLL2ARRAY(newaLL);
    end if
    validArrayLL.set(newaLL);
    arrayLLState.compareAndSet(rebuildInProgress, newaLL);
    rebuildGuard.compareAndSet(ThreadID, NOT_REBUILD);
end function
```

The optimized version of `rmFromArray()` should search for the item on the `arrayItem` and only when it finds a matching one, will check the node in the same index of `arrayNode` and finally, make a CAS operation on the `state` to logically remove the node. This version of `remove()` sets the `arrayLLState` variable to null upon successful completion, thus indicating to the next `contains()` operation that a rebuild is necessary.

6. Set Implementation

The CALL data structure can be modified to implement a *set* and in order to do so, the `add()` operation must be replaced with `addIfAbsent()`. Before adding a new item to the end of the list, the `addIfAbsent()` operation (described in Algorithm 7) must check if the item is already in the array, then check in the linked list, and then insert a node with the new item. Each time the CAS fails on the tail node, it must check if the newly inserted node contains the same item, and then try again, thus avoiding duplicates.

The `contains()` and `remove()` operations can be used as described in Algorithms 2 and 3.

Algorithm 7 Algorithm for `addIfAbsent()` operation

```
function ADDIFABSENT(item)
    aLL = validArrayLL.get();
    if FINDINARRAY(aLL, item) then
        return false;
    end if
    localTail = FINDINLLORTAIL(aLL, item);
    if localTail.item == item then
        if localTail.state.get() ≠ REMOVED then
            return false;
        end if
    end if
    newNode = new Node(item, INUSE);
    while true do
        if localTail == tail.get() then
            if localTail.next.get() == null then
                if localTail.next.CAS(null, newNode) then
                    tail.CAS(localTail, newNode);
                    return true;
                end if
            end if
            tail.CAS(localTail, localTail.next.get());
        end if
        localTail = localTail.next.get();
        if localTail.item == item then
            if localTail.state.get() ≠ REMOVED then
                return false;
            end if
        end if
    end while
end function
```

7. Performance Evaluation

A set of performance tests were conducted on a dual Opteron 6272 with a total of 32 cores, running Windows 7 Professional SP1, with JDK 8u45 64-bit.

We executed 5 individual runs for each of the data structures presented below, measuring the number of operations over a period of 20 seconds. Each data point in Figure 4 represents the median of the five runs for the total number of operations per second (reads+writes).

On every run, all data structures maintain the initial number of items: one thousand, ten thousand, and one million items. On each run, there were also six different workload configurations, 0%, 0.1%, 1%, 10%, 50%, and 100% writes, where each percentage value represents the probability that a write operation will be done, using a random number generator to determine whether it is a read or write operation. Every time a write operation occurs, it will randomly select an item from the possible items, add it to the data structure and immediately remove it. We do this because all the tested data structures are multisets, and as such, allow for multiple copies of the same item, but because we immediately remove it, the average number of items in the data structure remains constant. When a read operation is selected, it will do two calls to `contains()`, to match in number, the `remove()` and `add()` calls of the write operation.

- **CALLBase**: CALL basic algorithm described in section 3.1.
- **CALLItem**: CALL basic algorithm with the `arrayItem` optimization described in section 5.1.
- **CALL**: CALL algorithm with all the optimizations described in section 5: `arrayItem`, and add an `arrayLLState` that allows a *fast-path* for `contains()`.

- **CLQ**: ConcurrentLinkedQueue [8] from java.util.concurrent package, a lock-free linked list based on the algorithm described in [12].
- **COWArrayList**: CopyOnWriteArrayList [9] from the java.util.-concurrent package, an array based implementation using the COW pattern with wait-free `contains()`, and blocking write operations.

The results show that for most scenarios, the CALL algorithm provides the best performance. For all key ranges, under 1% writes, COWArrayList has a slight advantage over CALL, but as soon as the write ratio goes to 10% or higher, CALL provides better output. Regarding the CLQ, it only has better performance on lists with one thousand items and 50% or above of write operations. This happens because although CLQ and CALL use the same algorithm for `add()`, CLQ needs to go over a linked list to find the node to be removed, while CALL goes through an array for most of the time. Considering the plots with a key range of one million, CALL achieves a throughput up to 200 times higher than COWArrayList, and up to 30 times higher than CLQ, showing that CALL is best suited for large data sets. Curiously, there is an unexpected improvement on the number of operations when the percentage of write operations increase from 10% to 50%. We believe that this effect can be explained by a lower time of execution for the `add()` operation when compared to `rebuildAttempt()`, because it doesn't have to traverse the list, in conjunction with a `rebuildAttempt()` method that aggregates an higher number of mutations per rebuild.

It is fair to conclude that the CALL performs the best with an higher number of items in the *multiset*, with at least 10% write operations, which means that COWArrayList is only an alternative when there are few writes. When the ratio of write to read operations is unknown, or known to be volatile, the CALL algorithnm provides the best overall performance, because even with occasional writes, its performance is still close to the best data structure for that workload (the COWArrayList).

8. Conclusion

Unordered multisets and sets are commonly used data structures in concurrent applications. In this paper, we have presented a concurrent unordered multiset and unordered set implementations that are simple, non-blocking, practical, and fast when compared with similar data structures.

This algorithm needs a GC but it should be possible to modify it to work on runtimes without automatic garbage collection.

CALL is a lock-free data structure that is essentially based on an array, even if temporarily its linked list may contain items. In scenarios where insertions are bursty or rare, all of its items will be present in the array most of the time.

Performance measurements from our microbenchmark show that, although the CALL may underperform the competitor data structures for some workloads, like the COWArrayList for 0.1% writes, or the CLQ for 50% or 100% writes with 1000 items, for most scenarios it outperforms the other compared data structures, and it is always at least the second-best.

References

[1] N. N. Dang and P. Tsigas. Progress guarantees when composing lock-free objects. In *Euro-Par 2011 Parallel Processing*, pages 148–159. Springer, 2011.

[2] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. *PODC'04*, 2004.

[3] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[4] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems*, pages 3–16. Springer, 2006.

[5] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming - revised first edition*. Morgan Kaufmann, 2008.

[7] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[8] D. Lea. ConcurrentLinkedQueue. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`, 2013.

[9] D. Lea. CopyOnWriteArrayList. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html`, 2013.

[10] D. Lea. CopyOnWriteArraySet. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArraySet.html`, 2013.

[11] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[12] M. Scott and M. Michael. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *PODC*, 1996.

[13] B. Stroustrup. C++11 Style - A Touch of Class. `http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style`, 2012.

[14] H. Sutter. Modern C++: What you need to know. `http://channel9.msdn.com/Events/Build/2014/2-661`, 2014.

[15] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In *Principles of Distributed Systems*, pages 330–344. Springer, 2012.

[16] J. D. Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[17] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

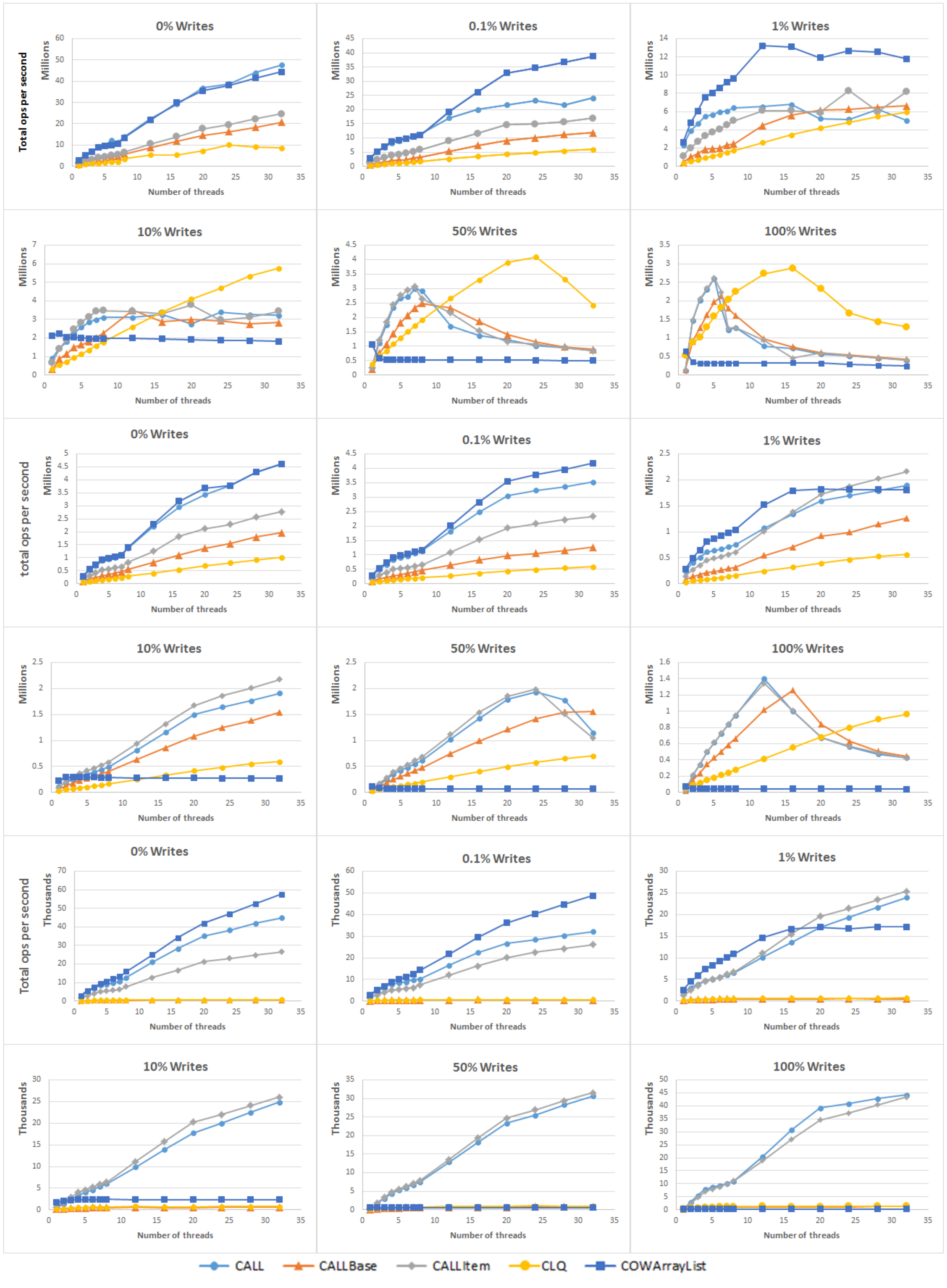

Figure 4. Performance comparison between multisets. Each two rows starting from the top represent a key range of 10^3, 10^4, and 10^6 respectively.