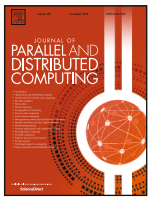
Journal of Parallel and Distributed Computing 211 (2026) 105238

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



New concurrent order maintenance data structure

Bin Guo [a,*], Emil Sekerinski [b]

[a] Department of Computer Science, Trent University, 1600 West Bank Drive, Peterborough, K9L 0G2, ON, Canada
[b] Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, L8S 4L8, ON, Canada

ARTICLE INFO

Keywords:
Order maintenance
Parallel
Multicore
Shared memory
Compare-and-swap
Lock-free
Amortized constant time
Core maintenance

ABSTRACT

The *Order-Maintenance* (OM) data structure maintains a total order list of items for insertions, deletions, and comparisons. As a basic data structure, OM has many applications, such as maintaining the topological order, k-core, and k-truss in graphs, and maintaining ordered sets in the Unified Modelling Language (UML) specification. The prevalence of multicore machines suggests parallelizing such a basic data structure. This paper proposes a new parallel OM data structure that supports insertions, deletions, and comparisons in parallel. Specifically, parallel insertions and deletions are efficiently synchronized using locks, which achieves up to 7x and 5.6x speedups with 64 workers. One significant advantage is that comparisons are lock-free, enabling them to execute highly in parallel with other insertions and deletions, which achieves up to 34.4x speedups with 64 workers. Typical real applications maintain order lists that always have a much larger portion of comparisons than insertions and deletions. For example, in core maintenance, the number of comparisons is up to 297 times larger compared with insertions and deletions in certain graphs. This shows that the lock-free order comparison provides a significant practical contribution.

1. Introduction

The well-known *Order-Maintenance* (OM) data structure [1–3] maintains a total order of unique items in an order list, denoted as $\mathbb{O}$, by following three operations:

- `Order`$(\mathbb{O}, x, y)$: determine if x precedes y in the ordered list $\mathbb{O}$, denoted as $x \preceq y$, supposing both x and y are in $\mathbb{O}$.
- `Insert`$(\mathbb{O}, x, y)$: insert a new item y after x in the ordered list $\mathbb{O}$, supposing that x is in $\mathbb{O}$ and y is not in $\mathbb{O}$.
- `Delete`$(\mathbb{O}, x)$: delete x from the ordered list $\mathbb{O}$, supposing that x is in $\mathbb{O}$.

Applications. The OM data structure is widely used for cohesive subgraph algorithms, including *k-core maintenance* [4–6] and k-truss maintenance [7], in *dynamic graphs*, where the graph has frequently inserted and removed edges. After computing the k-core and k-truss, they are maintained when updating edges instead of being recomputed inefficiently. The fundamental issue is that all vertices are maintained in a total order, the so-called *k-order*, which can be used to avoid repeated traversing of edges and thus improve performance. Similarly, the OM data structure can be used to maintain the *topological order* of vertices in directed acyclic graphs after dynamically inserting or removing edges [8,9]. Additionally, ordered sets are widely used in Unified Modeling Language (UML) Specifications [10], e.g., a display screen (an OS's representation) has a set of windows, but furthermore, the set is ordered, so are the ordered bag and sequence. In summary, the OM data structure is a building block for a batch of algorithms and is widely applied in extensive applications.

Sequential. In the sequential case, the OM data structure has been well studied. The naive idea is to use a balanced binary search tree [11]. All three operations can be performed in $O(\log N)$ time, where there are at most N items in the ordered list $\mathbb{O}$. In [1,2], the authors propose an OM data structure that supports all three operations in $O(1)$ time. The idea is that all items in $\mathbb{O}$ are linked as a double-linked list. Each item is assigned a label to indicate its order. We can perform the `Order` operation by comparing the labels of two items in $O(1)$ time. Also, the `Delete` operation costs $O(1)$ time without changing other labels. For the `Insert`(x, y) operation, y can be directly inserted after x with $O(1)$ time, if there exists label space between x and its successors; otherwise, a *relabel* procedure is triggered to rebalance the labels, which costs amortized $O(\log N)$ time per insertion. After introducing a list of sublists structure, the amortized running time of the relabel procedure can be optimized to $O(1)$ per insertion. Thus, the `Insert` operation has $O(1)$ amortized time.

Parallel. Due to the prevalence of the multicore shared-memory architecture, it immediately suggests parallelizing the OM data structure. In this paper, we present a new concurrent OM data structure. In terms of

* Corresponding author.
E-mail addresses: binguo@trentu.ca (B. Guo), emil@mcmaster.ca (E. Sekerinski).

https://doi.org/10.1016/j.jpdc.2026.105238
Received 11 October 2024; Received in revised form 30 January 2026; Accepted 8 February 2026
Available online 19 February 2026

Table 1
The worst-case and best-case work, depth complexities of parallel OM operations, where m is the number of operations executed in parallel, $\mathcal{P}$ is the total number of workers, and † is the amortized complexity.

Parallel	Worst-case (O)			Best-case (O)		
operation	$\mathcal{W}$	$\mathcal{D}$	time	$\mathcal{W}$	$\mathcal{D}$	time
`Insert`	$m^\dagger$	$m^\dagger$	$\frac{m}{\mathcal{P}} + m^\dagger$	$m^\dagger$	$1^\dagger$	$\frac{m}{\mathcal{P}}^\dagger$
`Delete`	m	m	$\frac{m}{\mathcal{P}} + m$	m	1	$\frac{m}{\mathcal{P}}$
`Order`	m	1	$\frac{m}{\mathcal{P}}$	m	1	$\frac{m}{\mathcal{P}}$

parallel `Insert` and `Delete` operations, we use locks for synchronization without allowing interleaving. In the average case, there is a high probability that multiple `Insert` or `Delete` operations occur in different positions of $\mathbb{O}$ so that these operations can execute completely in parallel. For the `Order` operation, we adopt a lock-free mechanism, which allows to run completely in parallel for any pair of items in $\mathbb{O}$. To implement the lock-free `Order`, we devise a new algorithm for the `Insert` operation that always maintains *Order Snapshot* for all items, even if many relabel procedures are triggered. Here, the Order Snapshot means that the labels of items indicate their order correctly. As `Insert` operations always maintain the Order Snapshot, we do not need to lock a pair of items when comparing their labels in parallel. In other words, lock-free `Order` operations are based on `Insert` operations that preserve the Order Snapshot.

We analyze our parallel OM operations in the work-depth model [11,12], where the work, denoted as $\mathcal{W}$, is the total number of operations that are used by the algorithm, and the depth, denoted as $\mathcal{D}$, is the longest length of sequential operations [13]. The expected running time is $O(\mathcal{W}/\mathcal{P} + \mathcal{D})$ by using $\mathcal{P}$ workers with load balancing among those. In particular, the work and depth terms are equivalent for sequential algorithms.

Table 1 compares the worst-case and best-case work and depth complexities for the three OM operations when running the m operations of the same kind in parallel. In the best case, all three operations have $O(m)$ work and $O(1)$ depth. However, `Insert` has worst-case $O(m)$ work and $O(m)$ depth; such a worst-case is easy to construct by inserting m items into the same position of $\mathbb{O}$, and thus all insertions are reduced to running sequentially. The `Delete` operation also has the worst-case $O(m)$ work and $O(m)$ depth; but this worst case only happens when all deletions cause a blocking chain, which has a very low probability. Especially, since the `Order` is lock-free, it always has $O(m)$ work and $O(1)$ depth in the worst and best cases. This is why `Order` operation can always run in parallel and has a great speedup for multicore machines. The lock-free `Order` operation is an important contribution of this work.

We conduct extensive experiments on a 64-core machine over a variety of test cases to evaluate the parallelism of the new parallel OM data structure. With 64 workers, our parallel `Insert` and `Delete` achieve up to 7x and 5.6x speedups; our parallel `Order` achieves up to 34.4x speedups.

The rest of this paper is organized as follows. The related work is in Section 2. The preliminaries are given in Section 3. Our parallel OM data structure is discussed in Section 4. We conduct experimental studies in Section 5 and conclude this work in Section 6.

2. Related work

In [14], Dietz proposes the first order data structure, with `Insert` and `Delete` having $O(\log n)$ amortized time and `Order` having $O(1)$ time. In [15], Tsakalidis uses BB[α] trees to improve the update bound to $O(\log n)$ and then to $O(1)$ amortized time. In [1], Dietz et al. propose the fastest order data structure, which has `Insert` in $O(1)$ amortized time, `Delete` in $O(1)$ time, and `Order` in $O(1)$ time. In [2], Bender et al. propose significantly simplified algorithms that match the bounds in [1].

A special case of OM is the *file maintenance* problem [1,2], which is to store n items in an array of size $O(n)$. File maintenance has four operations, i.e., insert, delete, scan-right (scan next k items starting from e), and scan-left (analogous to scan-right).

For the parallel or concurrent OM data structure, there exists little work [3,16] to the best of our knowledge. In [16], the order list is split into multiple parts and organized as a B-tree, which sacrifices the $O(1)$ time for three operations; also, the relevant nodes in the B-tree are locked for synchronization. In [3], a parallel OM data structure is proposed specifically for *series-parallel (SP)* maintenance, which identifies whether two accesses are logically independent. Several parallelism strategies are present for the OM data structure combined with SP maintenance. We apply the strategy of splitting a full group into our new parallel OM data structure.

3. Preliminaries

In this section, for the OM data structure, we revisit the detailed steps of the sequential version [1–3]. This is the background to discuss the parallel version in the next section.

The idea is that items in the total order are assigned labels to indicate the order. Typically, each label can be stored as an $O(\log N)$ bits integer, where N is the maximal number of items in $\mathbb{O}$. Assume it takes $O(1)$ time to compare two integers. The `Order` operation requires $O(1)$ time by comparing labels; also, the `Delete` operation requires $O(1)$ time since after deleting one item all other labels of items are not affected.

In terms of the `Insert` operation, efficient implementations provide $O(1)$ amortized time. First, a *two-level* data structure [3] is used. That is, each item is stored in the bottom-list, which contains a *group* of consecutive elements; each group is stored in top-list, which can contain $\Omega(\log N)$ items. Both the top-list and the bottom-list are organized as double-linked lists, and we use $x.pre$ and $x.next$ to denote the predecessor and successor of x, respectively. Second, each item x has a top-label $L^t(x)$, which equals to x's group label denoted as $L^t(x) = L(x.group)$, and bottom-label $L_b(x)$, which is x's label. Integer L^t is in the range $[0, N^2]$ and integer L_b in the range $[0, N]$.

Initially, there can be N' items in $\mathbb{O}$ ($N' \leq N$), which are contained in N' groups, separately. Each group is assigned a top-label L^t with an N gap between neighbours, e.g., groups $g_1, g_2, g_3, \ldots, g_{1000}$ can have top-labels $N, 2N, 3N, \ldots, 1000N$. Each item is assigned a bottom-label L_b as $(\lfloor N/2 \rfloor - 1)$, e.g., a single item x_1 in the group g_1 has a bottom-label $(\lfloor N/2 \rfloor - 1)$.

Definition 1. [Order Snapshot] The OM data structure maintains the Order Snapshot for x precedes y in the total order, denoted as $\forall x, y \in \mathbb{O} : x \preceq y \equiv L^t(x) < L^t(y) \ \vee \ (L^t(x) = L^t(y) \wedge L_b(x) < L_b(y))$

The OM data structure maintains the Order Snapshot defined in Definition 1. In other words, to determine the order of x and y, we first compare their top-labels (group labels) of x and y; if they are the same, we continually compare their bottom labels.

3.1. Insert

The operation `Insert`$(\mathbb{O}, x, y)$ is implemented by inserting y after x in x's bottom-list, assigning y the label $L_b(y) = L_b(x) + \lfloor (L_b(x.next) - L_b(x))/2 \rfloor$, and setting y in the same group as x with $y.group = x.group$ such that $L^t(y) = L^t(x)$. If $L_b(x.next) - L_b(x) < 2$, the x's group is *full*, which triggers a *relabel* operation. Otherwise, y can successfully obtain a new label, then the insertion is complete in $O(1)$ time. The relabel operation has two steps. First, the full group is split into many new groups, each of which contains at most $\frac{\log N}{2}$ items, and new labels L_b are uniformly assigned for items in new groups. Second, newly created groups are inserted into the top-list, if new group labels L^t can be assigned. Otherwise, we have to *rebalance* the group labels. That is, from the current group g, we continuously traverse the successors g' until $L(g') - L(g) > j^2$, where j is the number of traversed groups. Then, new

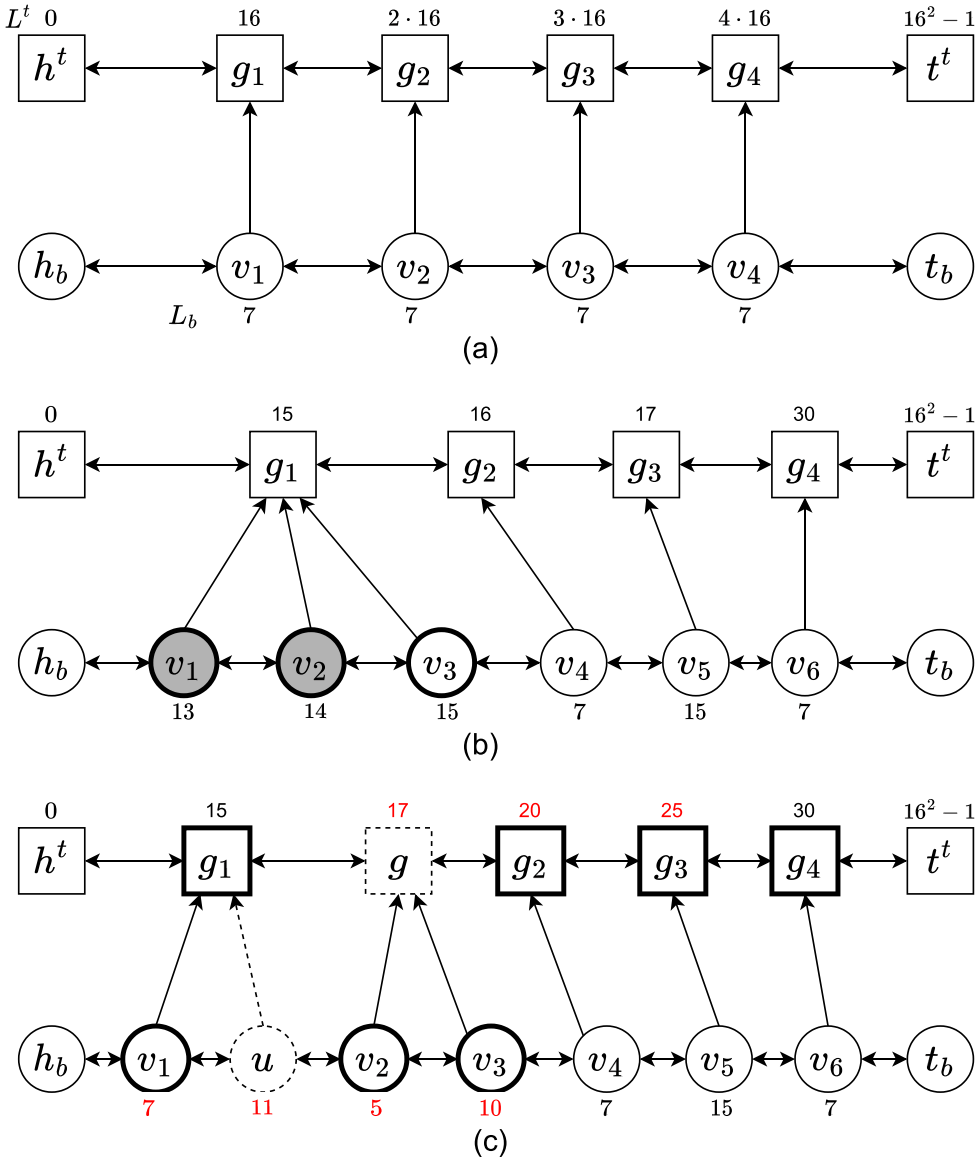


Fig. 1. An example of the OM data structure with $N = 16$. The squares are groups located in a single double-linked top-list with a head h^t and a tail t^t. The numbers at the top of the squares are group labels. The circles are items with pointers to their own groups, located in a single double-linked bottom-list. The numbers at the bottom of circles are item labels. (a) the initial state of $\mathbb{O} = \{v_1, v_2, v_3, v_4\}$. (b) the intermediate state of $\mathbb{O} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$. (c) after inserting a new item u (dashed circles) after v_1, we get $\mathbb{O} = \{v_1, u, v_2, v_3, v_4, v_5, v_6\}$ by inserting a new group g (dashed square).

group labels can be assigned to groups between g and g' with a $\frac{L(g')-L(g)}{j}$ gap, in which newly created groups can be inserted.

There are three important features in the implementation `Insert`: (1) each group, stored in the top-list, can contain $\Omega(\log N)$ items, so the total number of inserted groups is $\Omega(N/\log N)$ when inserting $\Omega(N)$ items; (2) the amortized cost of splitting groups is $\Omega(1)$ per insert; (3) the amortized cost of inserting a new group into the top-list is $\Omega(\log N)$ per insertion. Thus, each `Insert` operation only costs amortized $O(1)$ time.

Example 1. [Insert]

Fig. 1 shows a simple example of the OM data structure. For simplicity, we choose $N = 2^4 = 16$, so that for items the top-labels L^t are 8-bit integers (above groups as group labels), and the bottom labels are 4-bit integers (below items).

Fig. 1(a) shows an initial state of the two-level lists and labels. The top-list has head h^t and tail t^t labeled by 0 and $16^2 - 1$, respectively, and includes four groups g_1 to g_4 labeled with gap 16. The bottom-list has head h_b and tail t_b without labels, and includes four items v_1 to v_4 located in four groups g_1 to g_4 with the same labels 7.

Fig. 1(b) shows an intermediate state after a number of `Insert` and `Delete` operations. We can see that there does not exist a label space between v_1 and v_2. Both v_1 and v_2 are located in the group g_1. We get that g_1 is full when inserting a new item after v_1.

In Fig. 1(c), a new item u is inserted after v_1. But the group g_1 is full (no label space after v_1), which triggers a *relabel* process. That is, the group g_1 is split into two groups, g_1 and g; first, the old group g_1 has a single v_1, which can assign L_b, $15/2 = 7$; second, the newly created group g contains v_2 and v_3, which can assign L_b with a uniform distribution within their own group, $15/3 = 5$ and $2(15/3) = 10$, respectively. However, there is no label space between g_1 and g_2 to insert the new group g, which will trigger a *rebalance* operation. That is, we traverse groups from g_1 to g_4, where g_4 is the first that satisfies $L(g_4) - L(g_1) = 15 > j^2$ $(j = 3)$. Then, both g_2 and g_3 are assigned new

top-labels as 20 and 25, respectively, which have a gap of 5. Now, g can be inserted after g_1 with $L(g) = L(g_2) + (L(g_2) - L(g_1))/2 = 17$. Finally, we can insert u successfully after v_1 in g_1, with $L_b(u) = L_b(v_1) + (15 - L_b(v_1))/2 = 11$.

3.2. Atomic primitives

As shown in Algorithm 1, the `CAS` atomic primitive takes three arguments, a variable (location) x, an old value a and a new value b. It checks the value of the variable x, and if it equals to the old value a, it updates the pointer to the new value b and then returns *true*; otherwise, it returns *false* to indicate that the updating fails. Here, we use a pair of *angular brackets*, $\langle \ldots \rangle$, to indicate that the operations in between are executed atomically.

Algorithm 1: CAS(x, a, b).

```
1 ⟨ if x = a then x ← b; return true
2 else return false ⟩                    // ⟨...⟩ atomic
```

The modern multicore architectures support atomic primitives for reading or writing 64-bit integers. Typically, we assume that the order list $\mathbb{O}$ has at most 2^{32} items. In this case, the bottom-labels L_b are 32-bit integers and the top-labels L^t are 64-bit integers.

4. Parallel order maintenance data structure

In this section, we present the parallel version of the OM data structure. We start with the data structure definition and memory management. Then, we discuss the parallel `Delete` operations. Continually, we discuss the parallel `Insert` operation and show that the Order Snapshot is preserved at any steps, including the relabel process, which is the main contribution of this work. Finally, we present the parallel `Order` operation, which is lock-free and thus can be executed highly in parallel.

4.1. Data structure definition and memory management

We allocate two arrays in memory for one OM data structure. The first array `Items` stores all N items, and the second array `Groups` stores all N group nodes. Each item x has pointers that point to the previous item, the next item, and the parent group node. Similarly, each group node g has pointers that point to the previous group and the next group. All pointers are implemented using the array index rather than referencing a real memory address. Additionally, we need to allocate two arrays `ReclaimedItems` and `ReclaimedGroups` to store the reclaimed indices of items in `Items` and groups in `Groups` reclaimed (ready to be allocated to new items and groups), respectively. Algorithm 2 shows the detailed memory layout of the four allocated arrays.

An issue is how to manage the memory for `Delete` operations (described in Algorithm 3). Once an item x and a group g are deleted from `Items` and `Groups`, their index is added to `ReclaimedItems` and `ReclaimedGroups` without physically deallocating the memory, respectively. The index of the deleted item x and the deleted group g are added to `ReclaimedItems` and `ReclaimedGroups` for recycling, respectively. When a new item x and a group g are inserted, we can obtain a free item space and a free group space from `ReclaimedItems` and `ReclaimedGroups`, respectively.

This method only needs to allocate the memory once, thereby avoiding the frequent memory allocation for each inserted item or group. However, if there are more than N items or groups, all arrays need to be resized to obtain a larger capacity than before, by allocating new large arrays and copying the data from the old arrays to the new arrays. Each time, we can double the capacity of the arrays to reduce the number of memory copy operations.

Algorithm 2: Memory layout for items and groups stored.

```
1  Struct Item contains
2      pre: int
3      next: int
4      label: int with 32-bits
5      group: int
6      live: bool
7  Struct Group contains
8      pre: int
9      next: int
10     label: int with 64-bits
11     live: bool
12 Items ← new Item[N] as an array to store N items
13 Groups ← new Group[N] as an array to store N groups
14 ReclaimedItems ← new int[N] as an array to store the indices
     of reclaimed items
15 ReclaimedGroups ← new int[N] as an array to store the indices
     of reclaimed groups
```

4.2. Parallel delete

4.2.1. Algorithm

Algorithm 3 shows the detailed steps of parallel `Delete`. For each item x in $\mathbb{O}$, we use a Boolean status $x.live$ to indicate if x is in $\mathbb{O}$ or has been removed. Initially $x.live$ is `true`. We use the atomic primitive `CAS` to set $x.live$ from `true` to `false` to logically remove the item x first (line 1). The benefit of such deletion is that repeated deletion of x can be avoided by returning `failure`, and then x does not have to be locked for continuous operations. In lines 2–6 and 13, we remove x from the bottom-list in $\mathbb{O}$. To do this, we first lock $y = x.pre$, x, and $x.next$ in order to avoid deadlock (lines 2–4). Here, after locking y, we have to check that y still equals $x.pre$ in case $x.pre$ is changed by other workers (line 3). Then we can safely delete x from the bottom-list, and set x's pre, $next$, L_b, and $group$ to empty (lines 6 and 7). For the group $g = x.group$, we need to delete g when it is empty, which is analogous to deleting x (lines 8–14). We use the atomic primitive `CAS` to set $g.live$ from `true` to `false` to logically remove g from the list (line 8). Finally, we unlock all the locked items in reverse order (line 15).

Algorithm 3: Parallel-delete($\mathbb{O}, x$).

```
1  if not CAS(x.live, true, false) then return fail(   // logically
     delete x)
2  y ← x.pre; Lock(y)
3  if y ≠ x.pre then Unlock(y); goto line 2
4  Lock(x); Lock(x.next)
5  g ← x.group
6  delete x from bottom-list by setting x.pre.next ← x.next and
     x.next.pre ← x.pre
7  set x.pre, x.next, L_b(x), and x.group to ∅
8  if |g| = 0 ∧ CAS(g.live, true, false) then
9      g' ← g.pre; Lock(g')
10     if g' ≠ g.pre then Unlock(g'); goto line 9
11     Lock(g); Lock(g.next)
12     delete g from top-list by setting g.pre.next ← g.next and
         g.next.pre ← g.pre
13     set g.pre, g.next, and L(g) to ∅
14     Unlock(g.next); Unlock(g); Unlock(g')
15 Unlock(x.next); Unlock(x); Unlock(y);
```

4.2.2. Correctness

For deleting the item x, we lock three items, $x.pre$, x, and $x.next$ in order. Similarly, for deleting the group g, and lock $g.pre$, g and $g.next$ in order. Therefore, there are no blocking cycles and thereby no deadlock. After removing the item x from the bottom-list, the bottom-labels of all the other items in $\mathbb{O}$ are not changed. If the group g becomes empty after deleting x, g is also removed from the top-list, and the top-labels of all the other groups are not changed. Therefore the Order Snapshot is always preserved.

4.2.3. Complexities

Suppose there are m items to be deleted in the OM data structure. The total work is $O(m)$. In the best case, m items can be deleted in parallel by $\mathcal{P}$ workers with $O(1)$ depth, so that the total running time is $O(m/\mathcal{P})$. In the worst-case, m items have to be deleted one by one, e.g. $\mathcal{P}$ workers are blocked as a chain, with $O(m)$ depth, so that the total running time is $O(m/\mathcal{P} + m)$.

However, for deleting multiple items in parallel, a blocking chain is unlikely to appear when there is a large number of items with several workers. For example, given a list of three items x_1, x_2 and x_3, three workers p_1, p_2 and p_3 delete the three items, respectively. In the case of three workers running simultaneously, we have 1) p_3 first locks x_2 and x_3, 2) p_2 lock x_1 and wait to lock x_2 and x_3 (both already locked by p_3), and 3) p_1 waits to lock x_1 (already locked by p_2) and x_2 (already locked by p_3). We have p_1 waiting for p_2 and p_2 waiting for p_3, which forms a blocking chain and reduces to sequential. For another example, given a list of 1000 items $x_1, x_2, \ldots, x_{1000}$, three workers p_1, p_2 and p_3 delete the three items x_1, x_{100} and x_{200}, respectively. Three workers cannot form a blocking chain and can perform in a highly parallel way. In our experiments, there are millions of items and at most 64 workers, so the blocking chain of workers is unlikely to exist.

4.3. Parallel insert

4.3.1. Algorithm

Algorithm 4 shows the detailed steps for inserting y after x. Within this operation, we lock x and its successor $z = x.next$ in that order (lines 1 and 7). For obtaining a new bottom-label for y, if x and z are in the same group, $L_b(z)$ is the right bound; otherwise, N is the right bound, supposing L_b is a $(\log N)$-bit integer (line 2). If there does not exist a label gap in the bottom-list between x and $x.next$, we know that $x.group$ is full, and thus the `Relabel` procedure is triggered to make label space for y (line 3). Then, y is inserted into the bottom-list between x and $x.next$ (line 6), in the same group as x (line 4), by assigning a new bottom-label (line 5).

The `Relabel`(x) procedure splits the full group of x. We lock x's group g_0 and g_0's successor $g.next$ (line 9). We also lock all items $y \in g_0$ except x and $z = x.next$, as both x and z are already locked in line 1 (line 10). To split the group g_0 into multiple new smaller groups, we traverse the items $y \in g_0$ in reverse order by three steps (lines 11 - 15). First, if there does not exist a label gap in the top-list between g_0 and $g_0.next$, the `Rebalance` procedure is triggered to make label space for inserting a new group with assigned labels (lines 12 and 13). Second, we split $\frac{\log N}{2}$ items y from g_0 in reverse order to the new group g, which maintains the Order Snapshot (line 14). Third, we assign a new L_b to all items in the new group g by using the `AssignLabel` procedure (line 15), which also maintains the Order Snapshot. The for-loop (lines 11 - 15) stops if fewer than $\frac{\log N}{2}$ items are left in g_0. We assign a new L_b to all left items in g_0 by using the `AssignLabel` procedure (line 16). Finally, we unlock all locked groups and items (line 17).

In the `Rebalance`(g) procedure, we make label space after g to insert new groups. Starting from $g.next$, we traverse groups g' in order until $w > j^2$ by locking g' if necessary (g and $g.next$ are already locked in line 9), where j is the number of visited groups and w is the label gap $L(g') - L(g)$ (lines 19–22). That means j items will share a total of $w > j^2$ label space. All groups whose labels should be updated are added to the set A (line 21). We assign new labels to all groups in A by using the `AssignLabel` procedure (line 23), which maintains the Order Snapshot. Finally, we unlock groups locked in line 21 (line 24).

Notably, in the `AssignLabel`$(A, \mathcal{L}, l_0, w)$ procedure, we assign labels without affecting the Order Snapshot, where the set A includes all elements whose labels need updating, $\mathcal{L}$ is the label function, l_0 is the starting label, and w is the label space. Note that $\mathcal{L}$ can correctly return the bounded labels, e.g., $L_b(x.next) = N$ when x is at the tail of its group $x.group$. For each $z \in A$ in order, we first correctly assign a temporary label $\overline{\mathcal{L}}(z)$ (line 27), which can replace its real label $\mathcal{L}(z)$ at the right time by using the stack S (lines 28 - 32). Specifically, for each $z \in A$ in order, if its temporary label $\overline{\mathcal{L}}(z)$ is between $\mathcal{L}(z.pre)$ and $\mathcal{L}(z.next)$, we can safely *replace* its label by updating $\mathcal{L}(z)$ as $\overline{\mathcal{L}}(z)$ (lines 29 and 30), which maintains the Order Snapshot; otherwise, z is added to the stack S for further propagation (line 32). For propagation, when one element z replaces the labels (line 30), indicating that all elements in stack S can find enough label space, each $x \in S$ can be popped out by replacing its label (line 31). This propagation still maintains the Order Snapshot.

Example 2. [Parallel Insert] Fig. 1 shows an example for parallel `Insert`. In Fig. 1(b), we lock v_1 and v_2 in order when inserting u after v_1. However, there is no label space, and the group g_1 is full, which triggers the `Relabel` procedure. For the first step of relabelling, the other item v_3 in group g_1 is locked to split the group g_1.

In Fig. 1(c), we lock g_1 and g_2 in order when inserting a new group g after g_1, which triggers the `Rebalance` procedure on the top-list. For rebalancing, g_3 and g_4 are locked in order. The new temporary labels $\overline{L^t}$ of g_2 and g_3 are generated as $\overline{20}$ and $\overline{25}$. To replace real labels with temporary ones, we traverse g_2 and g_3 in order. First, we find that $L(g_1) < \overline{L}(g_2) < L(g_3)$ as $15 < \overline{20} < 17$ is false, so that g_2 is added to the stack such that $S = \{g_2\}$. Second, when traversing g_3, we find that $L(g_2) < \overline{L}(g_3) < L(g_4)$ as $16 < \overline{25} < 30$ is true, so that $L(g_3)$ is replaced as 25. In this case, the propagation of S begins, and g_2 is popped out with $L(g_2)$ replaced as 20. Finally, g_3 and g_4 are unlocked, and the `Rebalance` procedure finishes.

After rebalancing, the new group g can be inserted after g_1 with $L(g_1) = 17$. Relabeling continues. The item v_3 is split out to g with $L_b(v_3) = 15 \wedge L^t(v_3) = 17$ maintaining the Order Snapshot; similarly, v_2 is also split out to g. Now, both v_2 and v_3 require a new L_b to be assigned by the `AssignLabel` procedure. The new temporary label $\overline{L_b}$ of v_2 and v_3 are generated as $\overline{5}$ and $\overline{10}$, respectively. For replacement, we traverse v_2 and v_3 in order by two steps. First, for v_2, we find that $\overline{L_b}(v_2) < L_b(v_3)$ is true, so that $L_b(v_2)$ is replaced as 5. Second, for v_3, we find that $L_b(v_2) < \overline{L_b}(v_3)$ is true, so that $L_b(v_3)$ is replaced as 10 There is no further propagation since the stack S is empty. Now, only one item v_1 is left in g_1 and $L_b(v_1)$ is set to 7. Finally, the new item u is inserted after v_1 in g_1 with $L_b(u) = 11 \wedge L^t(u) = 15$.

Example 3 (Assign Label). In Fig. 2, we show an example of how the `AssignLabel` procedure preserves the Order Snapshot. The label space is from 0 to 15, shown as indices. There are four items v_1, v_2, v_3, and v_4 with initial labels 1, 2, 3, and 14, respectively; also, four temporary labels, 3, 6, 9, and 12, are assigned with uniform distribution to them. We traverse items from v_1 to v_4 in order. First, v_1 and v_2 are added to the stack S. Then, v_3 can safely replace its old label with its new temporary label 9, which makes space for v_2 that is at the top of S. So, we pop out v_2 from S and v_2 gets its new label 6, which makes space for v_1 that is at the top of S. So, we pop out v_1 from S and v_1 gets its new label 3. Finally, v_4 can safely get its new label 12. In a word, updating the v_3's label will repeatedly make space for v_2 and v_1 in the stack. During such a process, we observe that each time an old label is updated with a new temporary label, the labels always correctly indicate the order. Therefore, the Order Snapshot is always preserved and parallel `Order` operations can take place.

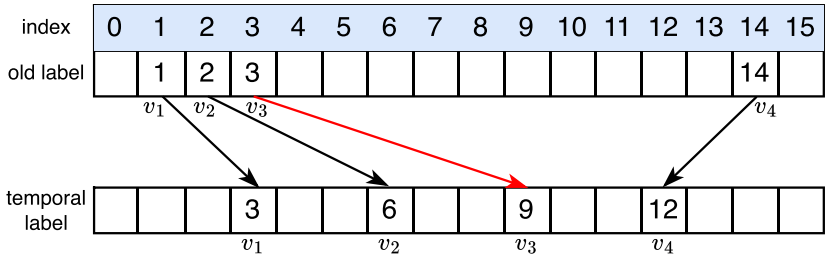


Fig. 2. An example of the `AssignLabel` procedure.

4.3.2. Correctness

There are four lock operations. First, we lock items x and $x.next$ in order (lines 1 and 7). Second, in the `Relabel` procedure, we lock g_0 and $g_0.next$ in order (lines 9 and 17). Third, we lock all items in the group g_0 except x and z that have been locked in line 1 (lines 9, 10, and 17) in order. Finally, we lock a batch of group g' in the while-loop in order (lines 21 and 24). We can see that all four locking operations lock items or groups in the order of their double-linked list from head to tail. Therefore, there are no blocking cycles and thereby no deadlock.

In Algorithm 4, there are two cases where the labels are updated: splitting groups (lines 11–15) and assigning labels (lines 15, 16, and 23) by using the `AssignLabel` procedure (lines 25–32). Intuitively, to prove that the Order Snapshot is preserved, we need to show that the labels can always correctly indicate the order at any time they are updating, including splitting groups and assigning labels. Specifically, we prove that the Order Snapshot is preserved with Theorems 1 and 2.

Theorem 1. *When splitting full groups (line 14), the Order Snapshot is preserved.*

Proof. The algorithm splits $\frac{\log N}{2}$ items y out from g_0 into the new group g (line 14), where each $y \in g_0$ is traversed in reverse order within the for-loop (lines 11–15). For this, the invariant of the for-loop is that y has largest L_b within g_0; the new group g has $L(g) > L(g_0)$; also, y satisfies the Order Snapshot:

$$(\forall x \in g_0 : x \neq y \implies L_b(y) > L_b(x))$$
$$\wedge\ (L(g_0) < L(g)) \wedge (y.pre \preceq y \preceq y.next)$$

We now argue that the for-loop preserves this invariant:

- $\forall x \in g_0 : x \neq y \implies L_b(y) > L_b(x)$ is preserved as y is traversed in reverse order within g_0 and all other items y' that have $L_b(y) < L_b(y')$ are already split out from g_0.
- $L(g_0) < L(g)$ is preserved as g is newly inserted into top-list after g_0.
- $y.pre \preceq y$ is preserved as we have $y \in g \wedge y.pre \in g_0$ and $L(g_0) < L(g)$.
- $y \preceq y.next$ is preserved as if y and $y.next$ all in the same group g, we have $L_b(y) < L_b(y.next)$; also, if y and $y.next$ in different groups, we have y is the first item moved to g or y is still located in g_0, which their groups indicates the correct order.

At the termination of the for-loop, the group g is split into multiple groups preserving the Order Snapshot. □

Theorem 2. *When assigning labels by using the `AssignLabel` procedure (lines 25–32), the Order Snapshot is preserved.*

Proof. The `AssignLabel` procedure (lines 25–32) assigns labels for all items $z \in A$. The temporary labels are first generated in advance (line 27). Then, the for-loop replaces the old label with new temporary labels (lines 28–32). The key issue is to argue the correctness of the inner while-loop (line 31). The invariant of this inner while-loop is that the top item in S has a temporary label that satisfies the Order Snapshot:

$$(\forall y \in S : (y \neq S.top \implies y \preceq S.top) \wedge y \preceq z)$$
$$\wedge\ x = S.top \implies \mathcal{L}(x.pre) < \overline{\mathcal{L}}(x) < \mathcal{L}(x.next)$$

The invariant initially holds as $\mathcal{L}(z)$ is correctly replaced by the temporary label $\overline{\mathcal{L}}(z)$ in line 30 and z is $x.next$, so that $\overline{\mathcal{L}}(x) < \mathcal{L}(z)$; also, we have $(x.pre) < \overline{\mathcal{L}}(x)$ as if it is not satisfied, x should not have been added into S, which causes contradiction. We now argue that the while-loop (line 31) preserves this invariant:

- $\forall y \in S : (y \neq S.top \implies y \preceq S.top) \wedge y \preceq z$ is preserved as all items in S are added in order, so the top item always has the largest order; also, since all item in A are traversed in order, so z has the larger order than all item in S.
- $x = S.top \implies \overline{\mathcal{L}}(x) < \mathcal{L}(x.next)$ is preserved as $\mathcal{L}(x.next)$ is already replace by the temporary label $\overline{\mathcal{L}}(x.next)$ and x is precede $x.next$ by using temporary labels.
- $x = S.top \implies \mathcal{L}(x.pre) < \overline{\mathcal{L}}(x)$ is preserved as if such an invariant is not satisfied, x should not be added into S, which causes a contradiction.

At the termination of the inner while-loop, we get $S = \emptyset$, so that all items that precede z have replaced new labels maintaining the Order Snapshot. At the termination of the for-loop (lines 28–32), all items in A have been replaced with new labels. □

4.3.3. Complexities

For the sequential version, it is proven that the amortized time is $O(1)$. The parallel version has some refinement. That is, the `AssignLabel` procedure traverses the locked items two times for generating temporary labels and replacing the labels, which costs amortized time $O(1)$. Thus, if m items are inserted in parallel, the total amortized work is $O(m)$. In the best case, m items can be inserted in parallel by $\mathcal{P}$ workers with amortized depth $O(1)$, so that the amortized running time is $O(m/\mathcal{P})$.

The worst-case can easily happen when all insertions are accrued in the same position of $\mathbb{O}$. The relabel procedure is triggered with the constant amortized work $\mathcal{W} = O(1)$ for each inserted item. In the worst-case, m items have to be inserted one-by-one, e.g. $\mathcal{P}$ workers simultaneously insert items at the head of $\mathbb{O}$ with amortized depth $O(m)$, and thus the amortized running time is $O(m/\mathcal{P} + m)$.

Such a worst-case can be improved by batch insertion. The idea is that we first allocate enough label space for $m/\mathcal{P}$ items per worker, then $\mathcal{P}$ workers can insert items in parallel. However, this simple strategy requires pre-processing of $\mathbb{O}$ and does not change the worst-case time complexity.

4.4. Parallel order

4.4.1. Algorithm

Algorithm 5 shows the detailed steps of `Order`. When comparing the order of x and y, they must not have been deleted (line 1). We first compare the top-labels of x and y (lines 2–5). Two variables, t and t', obtain the values of $L^t(x)$ and $L^t(y)$ for comparison (line 2), and the result is stored as r. After that, we have to check whether $L^t(x)$ or $L^t(y)$ has been updated or not; if that is the case, we have to *redo* the whole procedure (line 5), which is to go back to line 1 and execute lines 1 to 5 again. In other words, when comparing $L^t(x)$ and $L^t(y)$, both values cannot be updated by other workers. Second, we compare the bottom-labels of x

Algorithm 4: Parallel-insert($\mathbb{O}, x, y$).

```
1  Lock(x); z ← x.next; Lock(z)
2  if x.group = z.group then b ← L_b(z) else b ← N
3  if b − L_b(x) < 2 then Relabel(x, z)          // A full group triggers relabel
4  insert y into bottom-list between x and x.next by setting
     y.next ← x.next, y.pre ← x, x.next ← y, and x.next.pre ← y
5  L_b(y) ← L_b(x) + ⌊(b − L_b(x))/2⌋
6  y.group ← x.group
7  Unlock(x); Unlock(z)

8  procedure Relabel(x, z)
9      g_0 ← x.group; Lock(g_0); Lock(g_0.next);
10     Lock all items y ∈ g_0 with y ≠ x ∧ y ≠ z in order from head to tail
       // Split a full group
11     for y ∈ g_0 in reverse order until less than (log N)/2 items left in g_0 do
12         if L(g_0.next) − L(g_0) < 2 then Rebalance(g_0)
             // Rebalance groups
13         insert a new group g into the top-list after g_0 by setting
             g.next ← g_0.next, g.pre ← g_0, g_0.next ← g and
             g_0.next.pre ← g;  L(g) ←
             L(g_0) + ⌊(L(g_0.next) − L(g_0))/2⌋
14         split out (log N)/2 items y into g
15         AssignLabel(g, L_b, 0, N)
16     AssignLabel(g_0, L_b, 0, N)
17     Unlock all items y ∈ g_0 with y ≠ x ∧ y ≠ z, g_0.next, and g_0

18 procedure Rebalance(g)
19     g′ ← g.next; j ← 1; w ← L(g′) − L(g); A ← ∅
20     while w ≤ j^2 do
21         A ← A ∪ {g′}; g′ ← g′.next; Lock(g′)
22         j ← j + 1; w ← L(g′) − L(g)
23     AssignLabel(A, L^t, L^t(g), w)          // Assign labels for relabel process
24     Unlock all locked groups in line 21.

25 procedure AssignLabel(A, ℒ, l_0, w)
26     S ← empty stack; k ← 1; j ← |A| + 1
27     for z ∈ A in order do ℒ̄(z) = l + k · w/j; k ← k + 1
28     for z ∈ A in order do
29         if ℒ(z.pre) < ℒ̄(z) < ℒ(z.next) then
30             ℒ(z) ← ℒ̄(z)
31             while S ≠ ∅ do  x ← S.pop(); ℒ(x) ← ℒ̄(x)
32         else  S.push(z)
```

Algorithm 5: Parallel-order($\mathbb{O}, x, y$).

```
1  if x.live = false ∨ y.live = false then return fail
2  t, t′, r ← L^t(x), L^t(y), ∅
3  if t ≠ t′ then
4      r ← t < t′
5      if t ≠ L^t(x) ∨ t′ ≠ L^t(y) then goto line 1
6  else
7      b, b′ ← L_b(x), L_b(y); r ← b < b′
8      if t ≠ L^t(x) ∨ t′ ≠ L^t(y) ∨ b ≠ L_b(x) ∨ b′ ≠ L_b(y) then
9          goto line 1
10 if x.live = false ∨ y.live = false then return fail
11 return r
```

and y, if their top-labels are equal (lines 6–9). Similarly, two variables, b and b', obtain the value of $L_b(x)$ and $L_b(y)$ for comparison (line 7), and the result is stored as r. After that, we have to check whether four labels are updated or not; if any label is the case, we have to *redo* the whole procedure (lines 8 and 9). We can see that our parallel `Order` is lock-free, allowing for high parallelism. During the order comparison, x or y cannot be deleted (line 10). We return the result at line 11.

It is true that there is an *ABA problem*. That is, $L^t(x)$ and $L^t(y)$ are possibly updated multiple times but remain the same values as t and t' (line 5). In other words, $L^t(x)$ and $L^t(y)$ are updated but may not be identified when comparing t and t' (line 4), which can lead to a wrong result. Also, line 8 has the same problem. To solve this problem, each top-label or bottom-label, L^t or L_b, includes an 8-bit counter to record the version. Each time, the counter increases by one once its corresponding label is updated. With this implementation, we can safely check whether the label is updated or not merely by comparing the values (lines 5 and 8).

Example 4. [Order] In Fig. 2, we show an example to determine the order of v_2 and v_3 by comparing their labels. Initially, both v_2 and v_3 have old labels, 2 and 3. After the `Relabel` procedure is triggered, both v_2 and v_3 have new labels, 6 and 9, in which the Order Snapshot is preserved. However, it is possible that `Relabel` procedures are triggered in parallel. We first get $\mathcal{L}(v_3) = 3$ (old label) and second get $\mathcal{L}(v_2) = 6$ (new label), but it is incorrect for $\mathcal{L}(v_2) > \mathcal{L}(v_3)$. After we get $\mathcal{L}(v_2) = 6$, the value of $\mathcal{L}(v_2)$ has to be already updated to 9 since the Order Snapshot is maintained. In this case, we redo the whole process until $\mathcal{L}(v_2)$ and $\mathcal{L}(v_3)$ are not updated during comparison. Thus, we can get the correct result of $\mathcal{L}(v_2) < \mathcal{L}(v_3)$ even the relabel procedure is executed in parallel.

4.4.2. Correctness

We have proven that `Parallel-Insert` preserves the Order Snapshot even though relabel procedures are triggered, by which labels correctly indicate the order. In this case, it is safe to determine the order for x and y in parallel. We first argue the top-labels (lines 2 - 5). The problem is that we first get $t \leftarrow L^t(x)$ and second get $t' \leftarrow L^t(y)$ successively (line 2), by which l and l' may be inconsistent, due to a `Relabel` procedure may be triggered. To argue the consistency of labels, there are two cases: 1) both t and t' obtain old labels or new labels, which can correctly indicate the order; 2) the t first obtains an old label and t' second obtains a new label, which may not correctly indicate the order as x may already updated with a new label, and vice versa; if that is the case, we redo the whole process. On the termination of parallel `Order`, the invariant is that t and t' are consistent and thus correctly indicate the order. The bottom-labels are analogous (lines 6–9).

We can see our parallel `Order` operation is lock-free. First, it does not use locks. Second, it is possible that the loops occur (lines 5 and 9), when other workers are doing the `Relabel` operation simultaneously, which causes the labels of related items to be updated. However, the `Relabel` has a low probability of happening (evaluated in Section 6.2), will be completed quickly, and thus the tow loops cannot all spin forever without successful. Third, our parallel `Order` is linearizable regardless of the loops that occur (lines 5 and 9). Therefore, our parallel `Order` operation completes in a finite number of steps and has lock-freedom.

4.4.3. Complexities

For the sequential version, the running time is $O(1)$. For the parallel version, we have to consider the frequency of redo. It has a significantly low probability that the redo will be triggered. This is because the labels are changed by the `Relabel` procedure, which is triggered when inserting $\Omega(\log N)$ items. Even if the labels of x and y are updated when comparing their order, it still has a tiny probability that such label updating happens during the comparison of labels (lines 4 and 7).

Thus, supposing m items are comparing their orders in parallel, the total work is $O(m)$, and the depth is $O(1)$ with a high probability. So that the running time is $O(m/P)$ with high probability.

5. Implementation

In this section, we discuss the implementation details of our method.

5.1. Lock implementation

OpenMP (Open Multi-Processing) [17] is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures, and operating systems. We use OpenMP as the threading library to implement the parallel algorithms. In this work, the key issue is how to implement the synchronization locks. There are two different locks. One solution is to use the OpenMP lock, "`omp_set_lock`" and "`omp_unset_lock`". Each worker will suspend the working task until the specified lock is available. The OpenMP lock is efficient when substantial computation is performed within the locked region with few lock and unlock operations. The reason is that suspended workers require a low cost, but the operations of suspending and waking up workers have a high cost.

The other solution is the spin lock, which can be implemented by the atomic primitive `CAS`. Given a variable x as a lock, the `CAS` will repeatedly check x, and set x from `false` to `true` if x is `false`. In other words, one worker will busy-wait for the lock x until it is released by other workers without suspension.

Algorithm 6 shows an implementation of the spin lock. To reduce bus traffic, $x.lock$ is tested before using `CAS` to set $x.lock$ from `false` to `true` (line 3). Furthermore, it is more effective for other workers to *back-off* for some duration, giving competing workers the opportunity to acquire the lock. Typically, especially in our use cases, the large number of unsuccessful attempts indicates that the worker should back off for a longer period. Here, we use a simple strategy that exponentially increases the back-off time for each try (lines 1 and 4–6), where i and j are local variables without increasing bus traffic [18].

Algorithm 6: Lock(x).

```
1 i ← 1
2 while true do
3     if x.lock = false ∧ CAS(x.lock, false, true) then return
4     j ← i
5     while j > 0 do j ← j − 1
6     i ← 2 × i
```

5.2. Capacity of OM

In our experiment, for easy implementation, the bottom-labels L_b can be 32-bit integers, and the top-labels L^t can be 64-bit integers. One advantage is that reading and writing such 32-bit or 64-bit integers are atomic operations on modern machines. As we use 8-bit for the version number to avoid the ABA problem in Section 4.4.1. So, the total capacity of N is $2^{32-8} = 2^{24}$, which is more than 16 million.

Currently, both new modern ARM and X86 architectures already support 128-bit atomic `CAS` operation. Other atomic operations, such as `Read` and `Write`, do not support 128-bit atomicity, but we can emulate them using the `CAS` operation. So, we can get a large capacity up to $2^{64-8} = 2^{56}$.

6. Experiments

In this section, we implement our parallel approach, the so-called "*Ours*". We also implement the existing sequential approach in [2] as a baseline for comparison, the so-called "*Seq*". Note that *Seq* has almost the same performance as *Ours* when executing with a single worker sequentially, and we cannot figure out the difference in the figures. Therefore, we report both `Seq` and `Ours` with a single worker in the same figure. Unfortunately, we are unable to implement the parallel OM data structure in [3], which is combined with series-parallel maintenance and has a totally different mechanism from our implementation. Thus, we do not experimentally compare this approach with ours.

For the existing parallel or concurrent OM data structure, the work in [16] is not published, and we cannot find the implementation; the work in [3] is proposed specifically for series-parallel (SP) maintenance, not for the commonly used OM data structure, and we cannot implement it for evaluation. Therefore, we cannot compare "*Ours*" with the above two methods.

Specifically, we evaluate three order maintenance operations, `Order`, `Insert`, and `Delete`. We have four test cases, *No*, *Few*, *Many*, and *Max*, for the number of triggered relabeling processes. The source code is available on GitHub.[1]

6.1. Experiment setup

The experiments are performed on a server with an AMD Ryzen Threadripper PRO 3995WX (Zen 2, 64 cores, 128 hyperthreads, 256 MB of last-level shared cache), which consists of multiple NUMA (Non-Uniform Memory Access) domains due to its chiplet-based architecture. The hyperthread technique allows the CPU to process two sets of instructions (threads) simultaneously on one core by utilizing "dead time" when the core would otherwise be waiting for data. We choose the number of workers to increase exponentially, as $1, 2, 4, 8, 16, 32, 64$, and 128, to evaluate the parallelism. For this purpose, we assign each worker as a working thread to be pinned to one CPU core using "`pthread_setaffinity_np`" supported by the Linux system, so that multiple workers can physically run in parallel. Threads were pinned in a NUMA-aware manner, filling all cores within a NUMA node before moving to the next node. Memory allocation followed the same NUMA locality policy to minimize remote memory access. With different numbers of workers, we perform each experiment at least 100 times and calculate the mean with 95% confidence intervals. The server has 256 GB memory and runs the Ubuntu Linux (22.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 11.2.0 with the -O3 option. OpenMP[2] version 4.5 is used as the threading library.

We evaluate the OM data structure with four experiments:

- *Insert*: we insert 10 million items into $\mathbb{O}$.
- *Order*: we compare the order of two randomly chosen items. There are 10 million `Order` operations in total.
- *Delete*: we delete all inserted items, a total of 10 million times.
- *Mixed*: again, we insert 10 million items, mixed with 100 million `Order` operations. For each inserted item, we compare its order with the other 10 items that are randomly chosen; therefore, there are a total of 100 million order comparisons. The purpose is to investigate the frequency of "redo" occurring in the `Order` operations when there are parallel `Insert` operations. This experiment is to simulate the OM operations in graph algorithms, e.g., k-core maintenance. Real data graphs, such as social networks, are typically sparse, with a number of edges that is approximately 10 times the number of vertices; traversing edges requires `Order` operations and dealing with vertices needs `Insert` or `Delete` operations.

For each experiment, we have four test cases by choosing different numbers of positions for inserting:

[1] https://github.com/Itisben/Parallel-OM.git
[2] https://www.openmp.org/

Table 2
The detailed numbers of the relabel procedure.

	Insert				*Mixed*
Case	Relabel#	L_b#	L^t#	AvgLabel#	OrderRedo#
No	0	10,000,000	0	1	0
Few	2483	10,069,551	4967	1	0
Many	356,624	19,985,472	5,754,501	2.6	0
Max	357,142	19,999,976	99,024,410	**11.8**	0

- *No Relabel* case: we have 10 million positions, the total number of initial items in $\mathbb{O}$, so that each position averagely has 1 inserted items. Thus, it almost has *no* `Relabel` procedures triggered when inserting.
- *Few Relabel* case: we randomly choose 1 million positions from 10 million items in $\mathbb{O}$, so that each position averagely has 10 inserted items. Thus, it is possible that a *few* `Relabel` procedures are triggered when inserting.
- *Many Relabel* case: we randomly choose 1000 positions from 10 million items in $\mathbb{O}$, so that each position averagely has 10,000 inserted items. Thus, it is possible that *many* `Relabel` procedures are triggered when inserting.
- *Max Relabel* case: we only choose a single position (at the middle of $\mathbb{O}$) to insert $10,000,000$ items. In this way, we obtain a *maximum* number of triggered relabel procedures.

All items are inserted on-the-fly without preprocessing. In other words, 10 million items are randomly assigned to multiple workers, e.g 32 workers, even if in the *Max* case all insertions are reduced to sequential execution.

6.2. Evaluating relabelling

In this test, we evaluate the `Relabel` procedure triggered by `Insert` operations over four test cases, *No*, *Few*, *Many*, and *Max*. Since the different numbers of workers of *Ours* exhibit the same trend, we have chosen 32 workers for this evaluation. Of course, both *Ours* and *Seq* have the same results, so we report them together.

In Table 2, columns 2–4 show the details in the *Insert* experiment, where *Relabel#* is the times of triggered `Relabel` procedures, L_b# is the number of updated bottom-labels for items, L^t# is the number of updated top-labels for items, and *AvgLabel#* is the average number of updated labels for each inserted items when inserting 10 million items. We can see that, for four cases, the amortized numbers of updated labels increase slowly, where the average numbers of inserted items for each position increase by 1, 10, 10,000, and 10 million. This is because our parallel `Insert` operations have $O(1)$ amortized work. Specifically, we make several observations:

- The *No* case does not trigger `Relabel`, updating only one L_b per insert.
- The *Few* case triggers 2.5 thousand `Relabel`, updating 1.007 L_b, 0.005 L^t, and totally about one label per inserted item.
- The *Many* case triggers 0.36 million `Relabel`, updating 2 L_b, 0.6 L^t, and totally about 2.6 labels per inserted item.
- The *Max* case triggers 0.36 million `Relabel`, which is the same as *Many* the case. But it updates 2 L_b, 9.9 L^t, totally about 11 labels per inserted item.
- The *Max* case is the worst-case of `Insert` operations, but it has acceptable about 11 updated labels for each inserted item. This is because the `Insert` operations have amortized $\mathbb{O}(1)$ time complexity. Although relabel processes introduce short bursts of work, they are unlikely to happen.

In Table 2, the last column shows the times of *redo* (the "`goto`" in lines 5 and 9 of Algorithm 5) for `Order` operations in the *Mixed* experiment, which are all zero. Since *Mixed* has mixed `Order` and `Insert` operations, we may redo the `Order` operation if the corresponding labels are being updated. However, `Relabel` happens with a low probability; also, it is a low probability that related labels are changed when comparing the order of two items. This is why the times of *redo* are zero, leading to high parallel performance.

6.3. Evaluating the running time

In this test, for *Ours*, we exponentially increase the number of workers from 1 to 128 and evaluate the real running time. For *Seq*, we use a single worker. We perform *Insert*, *Order*, *Delete*, and *Mixed* over four test cases, *No*, *Few*, *Many*, and *Max*.

The plots in Fig. 3 depict the performance. The x-axis is the number of workers, and the y-axis is the execution time (milliseconds). Note that we compare the performance by using two kinds of lock: the OpenMP lock (denoted as dashed lines) and the spin lock (denoted as solid lines), since both locks are widely used in concurrent programming. A first look reveals that running times normally decrease with increasing numbers of workers, except for the *Max* case over the *Insert* and *Mixed* experiments. Specifically, we make several observations:

- The *Seq* has the same performance as *Ours* when using spin locks and a single worker. Therefore, we depict the *Seq* together with *Ours* with a single worker in plots. The reason is that even *Ours* has extra cost on atomic `CAS` operations, there does not exist contention with a single worker, and thus these atomic operations can perform efficiently with significant optimization by the compiler.
- For the experiment of *Order*, it is implemented as lock-free without using OpenMP lock or spin lock. The *No* and *Few* cases are much slower than the *Many* and *Max* cases. The reason is that when each item is distributed averagely across the large size of the list for the *No* and *Few* cases, the comparisons are effectively random across a huge memory space, reducing cache hit rates. In contrast, the *Many* and *Max* cases repeatedly access a smaller subset of memory, improving cache locality and thereby the performance.
- Three experiments, *Insert*, *Delete*, and *Mixed*, which use the spin lock, are much faster than using the OpenMP lock. This is because the lock regions always have few operations, and busy waiting (spin lock) is much faster than suspension waiting (OpenMP lock). Unlike the above three experiments, the *Order* experiment does not show any differences since `Order` operations are lock-free without using locks for synchronization.
- For the *Max* case of *Insert* and *Mixed*, unexpectedly, the running times increase with increasing number of workers. The reason is that the `Insert` operations are reduced to sequential in *Max* case since all items are inserted in the same position. Thus, it has the highest contention on shared positions where multiple workers are accessing at the same time, especially for 64 workers.
- For the *Many* case of *Insert* and *Mixed*, the running times decrease until using 4 workers. From 8 workers, however, the running times begin to increase. This is because the `Insert` operations have only 1000 positions in the *Many* case, and thus it may have a high contention on shared positions when using more than 4 workers.
- Over the *Order* and *Delete* experiments, we can see that the *Many* and *Max* cases are always faster than the *Few* and cases. This is because the *Few* and *No* cases have $1,000$ and 1 operating positions, respectively; all of these positions can fit into the CPU cache with high probability, and accessing the cache is much faster than accessing the memory.
- Over the *Order* experiments, the *Many* and *Max* cases do not have expected scalability between 64-worker and 128-worker. The reason is that the CPU has only 64 hardware cores but supports 128 threads via hyperthreading, and a high number of workers leads to high contention, which affects parallel performance with 128 workers.

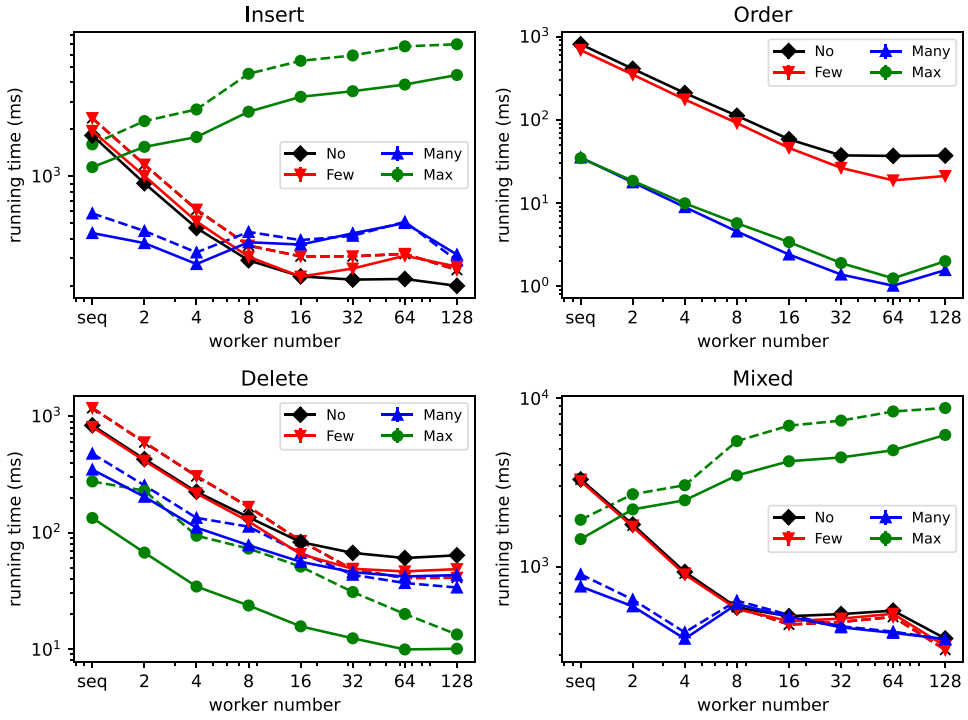


Fig. 3. Evaluating the running times.

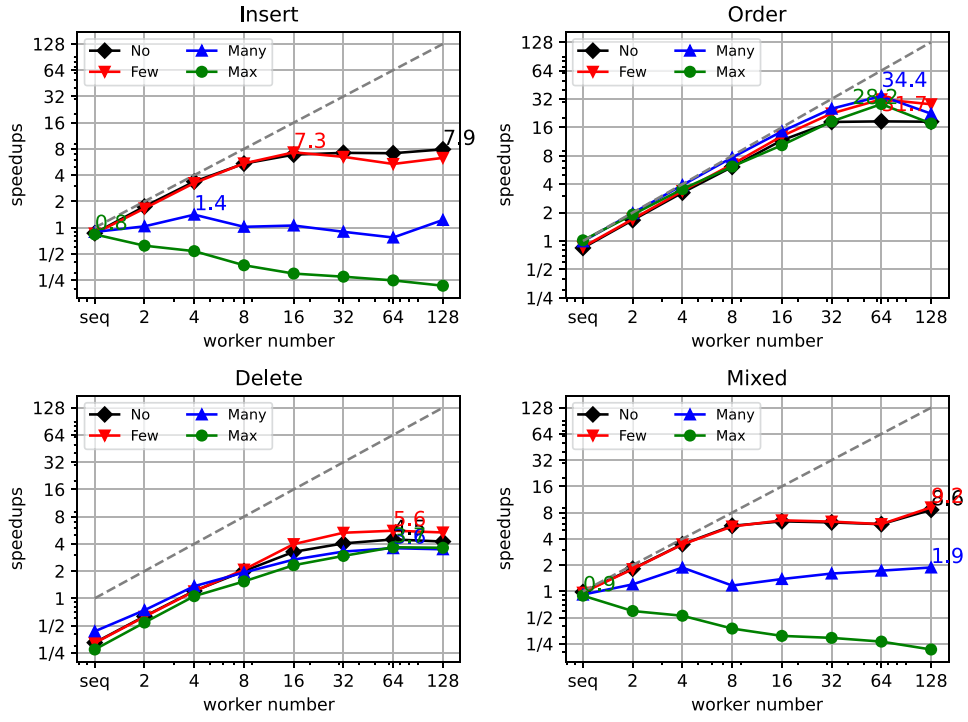


Fig. 4. Evaluate the speedups by increasing the number of workers (Strong Scaling).

6.4. *Evaluating the speedups*

In this experiment, we measure the Strong Scaling by increasing the number of workers and fixing the total work. The plots in Fig. 4 show the speedups of *Ours*. The x-axis is the number of workers, and the y-axis is the speedups, which are the ratio of running times (using spin locks) between the sequential version and using multiple workers. The dotted lines show the perfect speedups as a baseline. The numbers beside the lines indicate the maximal speedups. A first look reveals that all experiments achieve speedups when using multiple cores, except for the *Max* case over *insert* and *Mixed* experiments. Specifically, we make several observations:

– For all experiments, we observe that the speedups are around 1/4 to 1 when using 1 worker in all cases. This is because, for all operations of OM, the sequential version has the same work as the parallel version. Especially, for *Delete*, such speedups are low as 1/2 - 1/4, as locking items for deleting costs much running time.

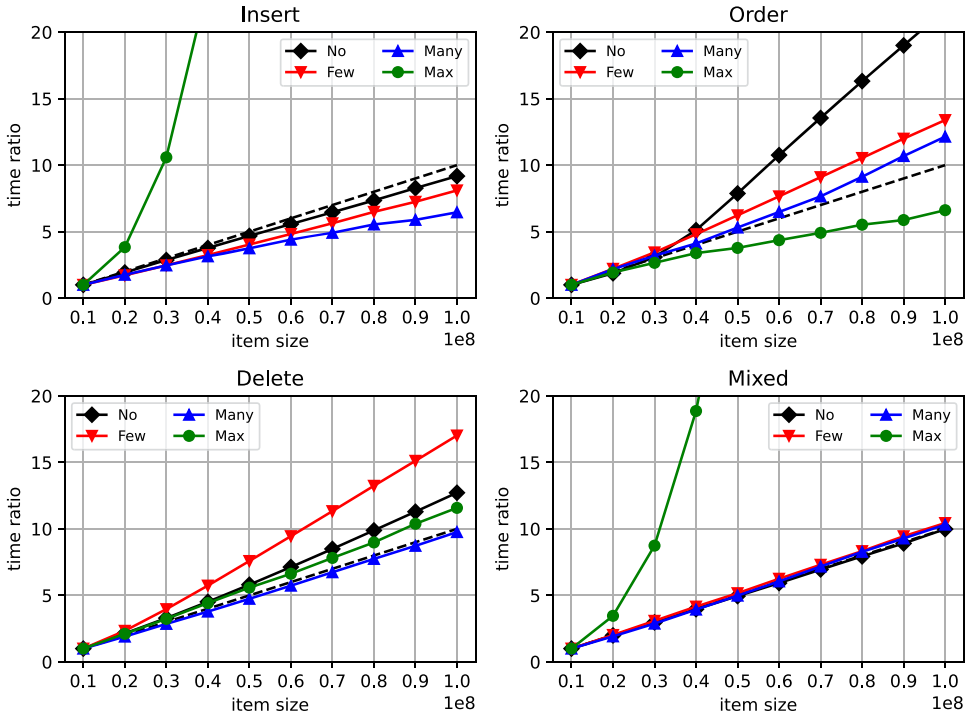


Fig. 5. Evaluate the scalability with 32 workers and increasing the item size (Weak Scaling).

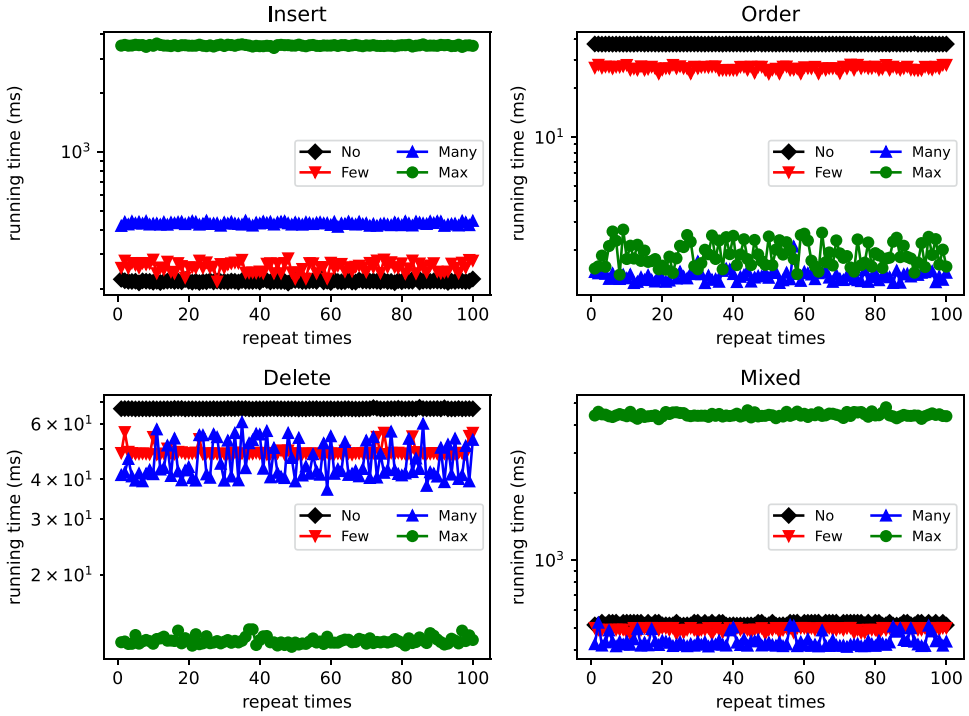


Fig. 6. Evaluating the stability of running times over 32 cores.

– For *Insert* and *Mixed*, we achieve around 7x speedups using 32 workers in *No* and *Few* cases, and around 2x speedups using 4 workers in *Many* cases. This is because all CPU cores have to access the shared memory through the bus, which connects memory and cores, and the atomic `CAS` operations will lock the bus. Each `Insert` operation may have many atomic `CAS` operations for spin lock and many atomic read and write operations for updating labels and lists. In this case, the bus traffic is high, which is the performance bottleneck for *Insert* operations.
– For *Order*, all four cases achieve almost perfect speedups from using 1 to 32 workers, as `Order` operations are lock-free.

– For *Delete*, it achieves around 4x speedups using 64 workers in four cases. This is because, for parallel `Delete` operations, the worst-case, where all operations are blocking as a chain, is almost impossible to happen.

6.5. Evaluating the scalability

In this experiment, we measure the Weak Scaling by increasing the total work and fixing with 32 workers . In this test of *Ours*, we increase the scale of the initial order list from 10 million to 100 million and evaluate running times with fixed 32 workers. We test three cases, *No*, *Few*, *Many*, and *Max*, by fixing the average number of items per insert position. For example, given an initial order list with 20 million items, the *No* case has 20 million insert positions, the *Few* case has 2 million positions, and the *Many* case has 2000 insert positions, and the *Max* case only has single one insert position.

The plots in Fig. 5 depict the performance. The x-axis is the initial size of the order list, and the y-axis is the time ratio of the current running time to the "10 million" running time. The dotted lines show the perfect time ratio as a baseline. The beginning time ratio is one. Obviously, we observe that for the parallel `Insert`, the time ratio of *Max* case increases sharply, since it is reduced to sequential and has no speedup with 32 workers, so does the *Mixed* case. Besides, all the other time ratios are roughly close to linearly increasing with the scales of the order list. This is because all parallel `Insert`, `Delete`, and `Order` have best-case time complexity $O(\frac{m}{P})$ and on average their running times are close to the best case.

Specifically, for `Order`, we can see that the time ratio is up to 20x with a scale of 100 million in the *No* case. The poor scalability of the *No* case workload for `Order` operations is due to the reduced cache locality, that is, each comparison touches random pairs of nodes distributed across the entire 10M array, which cannot fit into the L3 CPU cache anymore and may lead to cache misses. In contrast, the *Few*, *Many*, and *Max* cases concentrate accesses within fewer groups, which are more cache-friendly and thus more efficient than the *No* case.

6.6. Evaluating the stability

In this test of *Ours*, we compare 100 testing times for the *Insert*, *Order*, and *Delete* operations by using 32 workers. Each time, we randomly choose positions and randomly insert items for the *NO*, *Few*, and *Many* cases, so that the test is different. However, it is always the same for the *Max* case, since there is only one position to insert all items.

The plots in Fig. 6 depict the running time by performing the experiments 100 times. The x-axis is the index of repeating times, and the y-axis is the running times (milliseconds). We observe that the performance of *Insert*, *Order*, *Delete*, and *Mixed* remains well bounded across all four cases. Specifically, we have two observations:

– We can see that the *Max* case has a wider variation than other cases over *Insert* and *Mixed*. This is because the parallel `Insert` operations always have contention over shared data in memory. Such contention causes the running times to fluctuate within a bounded range.
– It is true that the relabel processes of `Insert` operations introduce short bursts of work. However, relabel processes are unlikely to happen since the groups are at least of size $\log N$ and the `Insert` operations have amortized $\mathbb{O}(1)$ time, the relabel cost is tightly bounded.

7. Conclusion and future work

We present a new parallel order maintenance (OM) data structure. The parallel `Insert` and `Delete` are synchronized with locks efficiently. Notably, the parallel `Order` is lock-free, and can execute highly in parallel. Experiments demonstrate significant speedups (for 64 workers) over the sequential version on a variety of test cases.

In future work, we will attempt to reduce the synchronization overhead, particularly for parallel `Insert`. Specifically, we will investigate the lock-free version of `Insert`, `Delete`, and `Order` operations by using the atomic Mult-Word Compare-and-Swap (MCAS) [19], which can significantly simplify the lock-free implementation. Furthermore, we will investigate insertions and deletions in batches by pre-processing the inserted and deleted items, which can significantly reduce the contention for multiple workers. In addition, we intend to apply our parallel OM data structure to a broad range of parallel algorithms.

CRediT authorship contribution statement

Bin Guo: Writing – review & editing, Writing – original draft; **Emil Sekerinski:** Supervision.

Data availability

No data was used for the research described in the article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] P. Dietz, D. Sleator, Two algorithms for maintaining order in a list, in: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, 1987, pp. 365–372.
[2] M.A. Bender, R. Cole, E.D. Demaine, M. Farach-Colton, J. Zito, Two simplified algorithms for maintaining order in a list, in: European Symposium on Algorithms, Springer, 2002, pp. 152–164.
[3] R. Utterback, K. Agrawal, J.T. Fineman, I.-T.A. Lee, Provably good and practically efficient parallel race detection for fork-join programs, in: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, 2016, pp. 83–94.
[4] Y. Zhang, J.X. Yu, Y. Zhang, L. Qin, A fast order-based approach for core maintenance, in: Proceedings - International Conference on Data Engineering, 2017, pp. 337–348. arXiv:1606.00200, https://doi.org/10.1109/ICDE.2017.93
[5] B. Guo, E. Sekerinski, Simplified algorithms for order-based core maintenance, (2022). arXiv:2201.07103
[6] B. Guo, E. Sekerinski, Parallel order-based core maintenance in dynamic graphs, arXiv:2210.14290 (2022).
[7] Y. Zhang, J.X. Yu, Unboundedness and efficiency of truss maintenance in evolving graphs, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1024–1041.
[8] M.A. Bender, J.T. Fineman, S. Gilbert, A new approach to incremental topological ordering, in: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2009, pp. 1108–1115.
[9] A. Marchetti-Spaccamela, U. Nanni, H. Rohnert, Maintaining a topological order under edge insertions, Inf. Process. Lett. 59 (1) (1996) 53–58.
[10] R.C. Martin, J. Newkirk, R.S. Koss, Agile Software Development: Principles, Patterns, and Practices, 2, Prentice Hall Upper Saddle River, NJ, 2003.
[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT press, 2022.
[12] J. Shun, Shared-Memory Parallelism Can be Simple, Fast, and Scalable, PUB7255 Association for Computing Machinery and Morgan & Claypool, 2017.
[13] J. JéJé, An Introduction to Parallel Algorithms, Reading, MA: Addison-Wesley, 1992.
[14] P.F. Dietz, Maintaining order in a linked list, in: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, 1982, pp. 122–127.
[15] A.K. Tsakalidis, Maintaining order in a generalized linked list, Acta Inf. 21 (1) (1984) 101–112.
[16] S. Gilbert, J. Fineman, M. Bender, Concurrent order maintenance, Unpublished (2003). https://ocw.mit.edu/courses/6-895-theory-of-parallel-systems-sma-5509-fall-2003/b5e012c63722c74d9d6504fef3caba00_gilbert.pdf.
[17] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel Programming in OpenMP, Morgan kaufmann, 2001.
[18] M. Herlihy, N. Shavit, V. Luchangco, M. Spear, The Art of Multiprocessor Programming, Newnes, 2020.
[19] R. Guerraoui, A. Kogan, V.J. Marathe, I. Zablotchi, Efficient multi-word compare and swap, in: 34th International Symposium on Distributed Computing, 2020.