New Parallel Order Maintenance Data Structure

Bin Guo*
Department of Computer Science
Trent University
Peterborough, Canada
binguo@trentu.ca

Emil Sekerinski
Department of Computing and Software
McMaster University
Hamilton, Canada
emil@mcmaster.ca

***Abstract*—The *Order-Maintenance* (OM) data structure is to keep a fully ordered list of items, supporting operations including insertions, deletions, and comparisons. As a crucial data structure, OM is widely used in various applications. With the increasing availability of multicore processors, there is a strong incentive to parallelize this essential data structure. This paper introduces a novel parallel OM data structure.**

***Index Terms*—order maintenance, parallel, multicore, lock-free**

I. Introduction

The widely used *Order-Maintenance* (OM) data structure [1]–[3] is responsible for preserving a total order of distinct elements within a list, denoted by $\mathcal{O}$, through the following three operations:

- ORDER(x, y): ascertain whether x precedes y in the ordered list $\mathcal{O}$, indicated as $x \preceq y$, assuming both x and y are in $\mathcal{O}$.
- INSERT(x, y): place a new item y immediately after x in the ordered list $\mathcal{O}$, provided x is in $\mathcal{O}$ and y is not.
- DELETE(x): remove x from the ordered list $\mathcal{O}$, assuming x exists in $\mathcal{O}$.

In this paper, we introduce a novel concurrent OM data structure. For parallel INSERT and DELETE operations, we utilize locks to synchronize the process, ensuring that no interleaving occurs. Typically, there is a high likelihood that multiple INSERT or DELETE operations will take place at different positions within $\mathcal{O}$, enabling these operations to execute entirely in parallel. For the ORDER operation, we employ a lock-free mechanism, allowing full parallel execution for any pair of items in $\mathcal{O}$. To achieve this, we develop a new algorithm for the INSERT operation that consistently maintains the *Order Snapshot* for all items, even when numerous RELABEL operations are initiated. The Order Snapshot ensures that item labels accurately reflect their order. Since INSERT operations consistently maintain the Order Snapshot, locking is not required when comparing item labels in parallel. Essentially, lock-free ORDER operations are supported by INSERT operations that preserve the Order Snapshot. The complete journal paper with all the details is available here[1].

II. Related Work

There is limited research on concurrent OM data structures. In [4], the order list is divided into several segments and structured as a B-tree, which compromises the $O(1)$ time complexity for the three operations; in addition, synchronization is achieved by locking the relevant nodes within the B-tree. In [3], a parallel OM data structure is introduced, particularly for *series-parallel (SP)* maintenance, which determines whether two accesses are logically independent. Various parallelism techniques are proposed for the OM data structure in conjunction with SP maintenance. We incorporate the method of splitting a complete group into our new parallel OM data structure.

III. Preliminaries

In this section, we review the detailed steps of the sequential version of the OM data structure [1]–[3], which sets the stage for discussing the parallel version in the following section.

The fundamental concept is that items in total order are assigned labels to represent their position. Typically, each label is stored as an integer of $O(\log N)$ bits, where N represents the maximum number of items in $\mathcal{O}$. It is assumed that comparing two integers takes $O(1)$ time. The ORDER operation is completed in $O(1)$ time by comparing the labels, and the DELETE operation also requires $O(1)$ time because the labels of the items are unaffected by the deletion of one item.

For the INSERT operation, efficient implementations provide an amortized time of $O(1)$.

- The first step involves using a *two-level* data structure [3]. In this structure, each item is stored in a Bottom-List containing a *group* of consecutive elements, while each group is stored in a Top-List, which can accommodate $\Omega(\log N)$ items. Both the Top-List and Bottom-List are structured as doubly linked lists, with $x.pre$ and $x.next$ denoting the predecessor and successor of x, respectively.
- Each item x also has a Top-Label $L^t(x)$, equivalent to its group's label, denoted as $L^t(x) = L(x.group)$, and a Bottom-Label $L_b(x)$, which is the label of x. The integer L^t ranges from $[0, N^2]$, while L_b ranges from $[0, N]$.

At the beginning, there can be N' items in $\mathcal{O}$ ($N' \leq N$), distributed among N' groups. Each group is assigned a Top-Label L with a gap of N between neighboring groups, and each item is assigned a Bottom-Label L_b of $\lfloor N/2 \rfloor$.

Definition III.1 (Order Snapshot)**.** An Order Snapshot is captured for the OM data structure, when x precedes y in the total order, defined as: $\forall x, y \in \mathcal{O} : x \preceq y \equiv L^t(x) < L^t(y) \vee (L^t(x) = L^t(y) \wedge L_b(x) < L_b(y))$

[1]https://arxiv.org/pdf/2208.07800

According to Definition III.1, the OM data structure indicates the Order Snapshot. That is, to determine the order of x and y, we first compare their Top-Labels (group labels); if these are identical, we then compare their bottom labels.

A. Sequential Insert Operation

The operation INSERT$(\mathcal{O}, x, y)$ is executed by placing y immediately after x in x's Bottom-List, assigning y the label $L_b(y) = \lfloor (L_b(x.next) - L_b(x))/2 \rfloor$, and ensuring y is in the same group as x, so that $L^t(y) = L^t(x)$. If $L_b(x.next) - L_b(x) > 1$, y successfully receives a new label, completing the insertion in $O(1)$ time. However, if the group containing x is *full*, a *RELABEL* operation is triggered.

The RELABEL operation involves two steps. The full group is first divided into multiple new groups, each containing at most $\frac{\log N}{2}$ items, and new labels L_b are uniformly assigned to items in these new groups. Then, the newly created groups are inserted into the Top-List if new group labels L^t can be assigned. Otherwise, a *REBALANCE* of group labels is required. This involves continuously traversing successors g' from the current group g until $L(g') - L(g) > j^2$, where j is the number of traversed groups. New group labels can then be assigned to groups between g and g', with a gap of $\frac{L(g')-L(g)}{j}$, allowing the insertion of the newly created groups.

The implementation of INSERT has three key points:

1) Each group, stored in the Top-List, contains $\Omega(\log N)$ items, resulting in a total number of insertions of $O(N/\log N)$.
2) The amortized cost of splitting groups is $O(1)$ per insertion.
3) The amortized cost of inserting a new group into the Top-List is $O(\log N)$ per insertion. Thus, each INSERT operation incurs an amortized cost of $O(1)$ time.

IV. Our Parallel Approach

We present the three operations of our parallel OM data structure in this section.

A. Parallel Delete

Removing an item x from the order $\mathcal{O}$ does not impact the labels of other items and consistently preserves the Order Snapshot. That is, the labels of the remaining items will continue to accurately reflect their order.

When removing x from $\mathcal{O}$, we lock $x.pre$, x, and $x.next$ sequentially to prevent deadlocks, and then safely remove x from the Bottom-List. The procedure is straightforward, so the detailed steps are omitted.

B. Parallel Insert

Algorithm 1 outlines the detailed steps for inserting y after x. During this process, x and its successor $z = x.next$ are locked sequentially (lines 1 and 7). To assign a new bottom label to y, if x and z belong to the same group, $L_b(z)$ is used as the upper bound; otherwise, N serves as the upper bound, assuming L_b is a $(\log N)$-bit integer (line 2). If there is no label gap in the Bottom-List between x and $x.next$, it indicates that $x.group$ is full, triggering the RELABEL operation to create label space for y (line 3). Subsequently, y is inserted into the Bottom-List between x and $x.next$ (line 6), in the same group as x (line 4), with a new bottom label assigned (line 5).

The RELABEL(x) procedure divides the full group of x. The group g_0 containing x and its successor $g.next$ are locked (line 9), along with all items $y \in g_0$, except x since x is already locked in line 1 (line 10). To divide g_0 into smaller groups, items $y \in g_0$ are traversed in reverse order through three steps (lines 11-15):

- First, if there is no label gap in the Top-List between g_0 and $g_0.next$, the REBALANCE procedure is invoked to create space for a new group with assigned labels (lines 12 and 13).
- Second, $\frac{\log N}{2}$ items y are moved from g_0 to the new group g, ensuring the Order Snapshot is maintained (line 14).
- Third, new labels L_b are assigned to all items in the new group g using the ASSIGNLABEL procedure (line 15), which also preserves the Order Snapshot. The for-loop (lines 11-15) halts if fewer than $\frac{\log N}{2}$ items remain in g_0, and new labels L_b are assigned to the remaining items in g_0 using the ASSIGNLABEL procedure (line 16).

Finally, all locked groups and items are unlocked (line 17).

In the REBALANCE(g) procedure, label space is created after g to accommodate new groups. Starting from $g.next$, groups g' are traversed until $w > j^2$, locking g' if necessary (g and $g.next$ are already locked in line 9), where j represents the number of traversed groups and w denotes the label gap $L(g') - L(g)$ (lines 19-22). This means that j items will share the $w > j^2$ label space. All groups requiring updated labels are added to set A (line 21). New labels are assigned to all groups in A using the ASSIGNLABEL procedure (line 23), maintaining the Order Snapshot. Finally, the groups locked in line 21 are unlocked (line 24).

In the ASSIGNLABEL$(A, \mathcal{L}, l_0, w)$ procedure, labels are assigned without disrupting the Order Snapshot. The set A includes all elements whose labels require updating, $\mathcal{L}$ is the label function, l_0 is the starting label, and w is the label space. Notably, $\mathcal{L}$ can correctly return bounded labels, such as $L_b(x.next) = N$ when x is at the end of its group $x.group$. For each $z \in A$ in order, a temporary label $\overline{\mathcal{L}}(z)$ is initially assigned (line 27), which can later replace its actual label $\mathcal{L}(z)$ using stack S (lines 28-32). Specifically, if the temporary label $\overline{\mathcal{L}}(z)$ for $z \in A$ falls between $\mathcal{L}(z.pre)$ and $\mathcal{L}(z.next)$, the label can be safely replaced by updating $\mathcal{L}(z)$ to $\overline{\mathcal{L}}(z)$ (lines 29 and 30), maintaining the Order Snapshot; otherwise, z is added to stack S for further propagation (line 32). During propagation, when the label of element z is replaced (line 30), all elements in stack S can find sufficient label space, and each $x \in S$ can be popped and its label replaced (line 31). This propagation also maintains the Order Snapshot.

a) Correctness.: All items and groups are locked sequentially, ensuring that there are no blocking cycles, which makes parallel insertions and deletions free from deadlocks.

```
1  LOCK($x$), $z \leftarrow x.next$, LOCK($z$)
2  if $x.group = z.group$ then $b \leftarrow L_b(z)$ else $b \leftarrow N$
3  if $b - L_b(x) < 2$ then RELABEL($x$)
4  inserting $y$ into Bottom-List after $x$ and before $x.next$
5  $L_b(y) \leftarrow L_b(x) + \lfloor (b - L_b(x))/2 \rfloor$
6  $y.group \leftarrow x.group$
7  UNLOCK($x$), UNLOCK($z$)
8  procedure RELABEL($x$)
9      $g_0 \leftarrow x.group$, LOCK($g_0$),LOCK($g_0.next$)
10     LOCK all items $y \in g_0$ with $y \neq x$ in order
11     for $y \in g_0$ in reverse order until less than $\frac{\log N}{2}$ items
         left in $g_0$ do
12         if $L(g_0.next) - L(g_0) < 2$ then REBALANCE($g_0$)
13         inserting a new group $g$ in the Top-List after $g_0$ with
             $L(g) = (L(g_0.next) - L(g_0))/2$
14         splitting out $\frac{\log N}{2}$ items $y$ into $g$
15         ASSIGNLABEL($g, L_b, 0, N$)
16     ASSIGNLABEL($g_0, L_b, 0, N$)
17     UNLOCK $g_0.next$, $g_0$, and all items $y \in g_0$ with $y \neq x$
18 procedure REBALANCE($g$)
19     $g' \leftarrow g.next$, $j \leftarrow 1$, $w \leftarrow L(g') - L(g)$, $A \leftarrow \emptyset$
20     while $w \leq j^2$ do
21         $A \leftarrow A \cup \{g'\}$, $g' \leftarrow g'.next$, LOCK($g'$)
22         $j \leftarrow j + 1$, $w \leftarrow L(g') - L(g)$
23     ASSIGNLABEL($A, L^t, L^t(g), w$)
24     UNLOCK all groups locked in line 21.
25 procedure ASSIGNLABEL($A, \mathcal{L}, l_0, w$)
26     $S \leftarrow$ empty stack, $k \leftarrow 1$, $j \leftarrow |A| + 1$
27     for $z \in A$ in order do $\overline{\mathcal{L}}(z) = l + k \cdot w/j$, $k \leftarrow k + 1$
28     for $z \in A$ in order do
29         if $\mathcal{L}(z.pre) < \overline{\mathcal{L}}(z) < \mathcal{L}(z.next)$ then
30             $\mathcal{L}(z) \leftarrow \overline{\mathcal{L}}(z)$
31             while $S \neq \emptyset$ do $x \leftarrow S.pop()$, $\mathcal{L}(x) \leftarrow \overline{\mathcal{L}}(x)$
32         else $S.push(z)$
```
Algorithm 1: Parallel-INSERT($\mathcal{O}, x, y$)

We demonstrate that the Order Snapshot is maintained during parallel INSERT operations. In Algorithm 1, there are two scenarios where labels are updated: during group splitting (lines 11-15) and during label assignment (lines 15, 16, and 23) using the ASSIGNLABEL procedure (lines 25-32).

Theorem IV.1. The Order Snapshot is preserved when full groups are split (line 14).

Proof. The algorithm separates $\frac{\log N}{2}$ items y from g_0 into the new group g (line 14), with each $y \in g_0$ traversed in reverse order within the for-loop (lines 11-15). The loop invariant is that y has the largest L_b within g_0; the new group g has $L(g) > L(g_0)$; additionally, y adheres to 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 show that the for-loop maintains this invariant:

- $\forall x \in g_0 : x \neq y \implies L_b(y) > L_b(x)$ is maintained because y is traversed in reverse order within g_0, and all other items y' with $L_b(y) < L_b(y')$ have already been split off from g_0.
- $L(g_0) < L(g)$ is preserved as g is newly inserted into the Top-List after g_0.
- $y.pre \preceq y$ is preserved because y is now in g while $y.pre$ remains in g_0, with $L(g_0) < L(g)$.
- $y \preceq y.next$ is preserved as if y and $y.next$ are both in the same group g, $L_b(y) < L_b(y.next)$; similarly, if y and $y.next$ are in different groups, y is either the first item moved to g or remains in g_0, with the groups correctly indicating the order.

At the conclusion of the for-loop, the group g is divided into multiple groups while preserving the Order Snapshot. □

Theorem IV.2. The Order Snapshot is maintained when assigning labels using the ASSIGNLABEL procedure (lines 25-32).

Proof. The ASSIGNLABEL procedure (lines 25-32) assigns labels to all items $z \in A$. Temporary labels are first generated (line 27), and then the for-loop replaces the old labels with these new temporary labels (lines 28-32). The critical point is to ensure 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 maintains 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 because $\mathcal{L}(z)$ is correctly replaced by the temporary label $\overline{\mathcal{L}}(z)$ in line 30, and since z is $x.next$, we have $\overline{\mathcal{L}}(x) < \mathcal{L}(z)$; also, $\mathcal{L}(x.pre) < \overline{\mathcal{L}}(x)$ holds because if it didn't, x would not have been added to S, leading to a contradiction. We now argue that the while-loop (line 31) maintains this invariant:

- $\forall y \in S : (y \neq S.top \implies y \preceq S.top) \wedge y \preceq z$ is preserved since all items in S are added in order, so the top item always has the highest order; additionally, since all items in A are traversed in sequence, z has a higher order than all items in S.
- $x = S.top \implies \overline{\mathcal{L}}(x) < \mathcal{L}(x.next)$ is preserved because $\mathcal{L}(x.next)$ is already replaced by the temporary label $\overline{\mathcal{L}}(x.next)$, and x precedes $x.next$ using the temporary labels.
- $x = S.top \implies \mathcal{L}(x.pre) < \overline{\mathcal{L}}(x)$ is maintained because if this invariant is not satisfied, x should not be added to S, and $\mathcal{L}(x)$ can be safely replaced by the temporary label, which would otherwise cause a contradiction.

When the inner while-loop ends, S is empty, meaning all items that precede z have had their labels replaced while maintaining the Order Snapshot. At the conclusion of the for-loop (lines 28-32), all items in A have been updated with new labels. □

b) Complexities.: For the sequential version, the amortized time has been shown to be $O(1)$ [1], [2]. The parallel version introduces some refinements. Specifically, the ASSIGNLABEL procedure involves traversing the locked items twice—first to generate temporary labels and then to replace them—which also incurs an amortized cost of $O(1)$ time. Therefore, when m items are inserted in parallel, the total amortized work remains $O(m)$.

```
1  if (not $x.live$) or (not $y.live$) then return FAIL
2  $t \leftarrow L^t(x), t' \leftarrow L^t(y), r \leftarrow$ FALSE
3  if $t \neq t'$ then
4      $r \leftarrow t < t'$
5      if $t \neq L^t(x)$ or $t' \neq L^t(y)$ then goto line 1
6  else
7      $b, b' \leftarrow L_b(x), L_b(y);\ r \leftarrow b < b'$
8      if $t \neq L^t(x)$ or $t' \neq L^t(y)$ or $b \neq L_b(x)$ or $b' \neq L_b(y)$
        then
9          goto line 1
10 if (not $x.live$) or (not $y.live$) then return FAIL
11 return $r$
```

Algorithm 2: Parallel-ORDER($\mathcal{O}, x, y$)

In the best-case scenario, m items can be inserted in parallel by $\mathcal{P}$ workers with an amortized depth of $O(1)$, resulting in an amortized running time of $O(m/\mathcal{P})$. However, in the worst-case scenario, where m items must be inserted sequentially (e.g., if $\mathcal{P}$ workers simultaneously insert items at the head of $\mathcal{O}$), the amortized depth becomes $O(m)$, leading to an amortized running time of $O(m/\mathcal{P} + m)$. This worst-case scenario occurs when all insertions happen at the same position in $\mathcal{O}$.

C. Parallel Order

Algorithm 2 details the steps for the `ORDER` operation. When comparing the order of x and y, it is crucial that neither has been deleted (line 1). The process begins by comparing the Top-Labels of x and y (lines 2-5). Two variables, t and t', are assigned the values of $L^t(x)$ and $L^t(y)$ for comparison (line 2), with the result stored in r. After this, it is necessary to verify whether $L^t(x)$ or $L^t(y)$ has been updated; if so, the entire procedure must be redone (line 5).

Next, if the Top-Labels of x and y are identical, the Bottom-Labels are compared (lines 6-9). Similarly, two variables, b and b', are used to retrieve the values of $L_b(x)$ and $L_b(y)$ for comparison (line 7), with the result again stored in r. It is then essential to check if any of the four labels have been updated; if so, the procedure must be repeated (lines 8 and 9). This parallel `ORDER` operation is lock-free, allowing it to execute highly in parallel.

During the order comparison, neither x nor y can be deleted (line 10), as the labels of deleted items would no longer accurately indicate order. The final result is returned at line 11.

An *ABA problem* exists in this context. Specifically, $L^t(x)$ and $L^t(y)$ might be updated multiple times but still hold the same values, t and t' (line 5). This means that while $L^t(x)$ and $L^t(y)$ may have been updated, these changes might not be detected when comparing t and t' (line 4), potentially leading to incorrect results. A similar issue arises in line 8. To address this problem, each Top-Label or Bottom-Label, L^t or L_b, can be implemented as a 64-bit integer, with a 32-bit counter embedded to track the version. Every time a label is updated, the corresponding counter increments by one. This approach allows for a reliable check to determine whether a label has been updated simply by comparing the values (lines 5 and 8).

a) Correctness.: We have demonstrated that the parallel `INSERT` operation preserves the Order Snapshot, even when RELABEL procedures are triggered, ensuring that labels continue to accurately reflect the order. This means it is safe to determine the order for x and y in parallel. Let's first consider the Top-Labels (lines 2-5). The issue arises because we first assign $t \leftarrow L^t(x)$ and then $t' \leftarrow L^t(y)$ in succession (line 2), which could lead to inconsistencies if a `RELABEL` procedure is triggered in the meantime. To establish label consistency, we consider two scenarios: 1) Both t and t' receive either old labels or new labels, which correctly indicate the order; 2) The t receives an old label while t' receives a new label, which could result in an incorrect order indication if x has already been updated with a new label, and vice versa. If this occurs, the entire process must be repeated. Upon completion of the parallel `ORDER`, the invariant is that t and t' are consistent and, therefore, accurately reflect the order. The same logic applies to the Bottom-Labels (lines 6-9).

b) Complexities.: For the sequential version, the running time is $O(1)$. In the parallel version, however, we need to account for the likelihood of a redo occurring. The chance of a redo being triggered is very low. This is because label changes occur due to the `RELABEL` procedure, which only happens when $\Omega(\log N)$ items are inserted. Even if the labels of x and y are updated during order comparison, the probability of this happening during the actual label comparison (lines 4 and 7) remains minimal. Therefore, when m items are compared in parallel, the total work remains $O(m)$, and the depth is $O(1)$ with high probability. Consequently, the running time is $O(m/\mathcal{P})$ with high probability.

V. Experiments

The experiments are conducted on a server equipped with an AMD CPU featuring 64 cores, 128 hyperthreads, and 256 MB of last-level cache, along with 256 GB main memory. The operating system is Linux (Ubuntu 22.04). All algorithms are implemented in C++ and compiled using g++ version 11.2.0 with the -O3 optimization flag. OpenMP[2] version 4.5 adopted for parallel computing. We increase the number of workers exponentially, with 1, 2, 4, 8, 16, 32, and 64 workers. Each experiment is repeated at least 100 times to obtain the mean (95% confidence intervals calculated). The source code is on GitHub[3].

For simplicity, we set $N = 2^{32}$, a 32-bit integer, as the capacity of $\mathcal{O}$. The Bottom-Labels L_b are 32-bit integers, and the Top-Labels L^t are 64-bit integers. One advantage of this setup is that reading and writing these 32-bit or 64-bit integers is atomic on modern hardware. Initially, the order list $\mathcal{O}$ contains 10 million items. To evaluate our parallel OM data structure, we conducted four types of experiments:

- *INSERT*: 10 million items are inserted into $\mathcal{O}$.
- *ORDER*: For each inserted item, its order is compared with its successive item, resulting in 10 million `ORDER` operations.

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

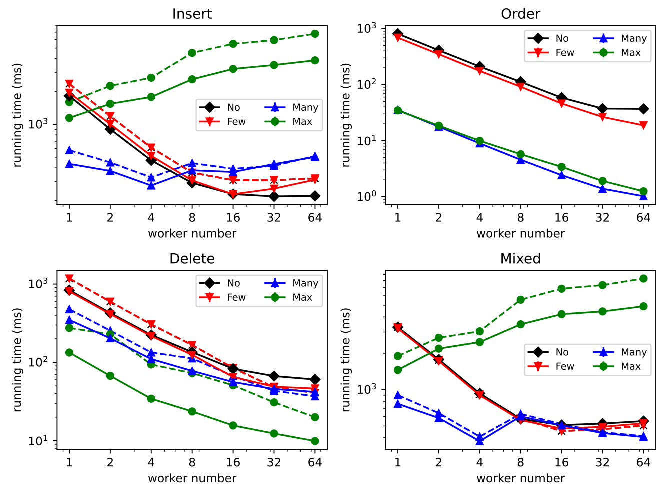

Fig. 1: The running times for *NO, FEW, MANY*, and *MAX* cases. We have that x-axis shows the number of workers, and the y-axis displays the execution time (in milliseconds), both on an exponential scale.

- *DELETE*: All inserted items are deleted, totally 10 million deletions.
- *MIXED*: 10 million items are inserted, mixed with 100 million `ORDER` operations. For each inserted item, its order is compared with the next ten items, 100 million order comparisons in total.

For each experiment, we considered four test cases based on the number of insertion positions:

- *NO*: 10 million positions are used, so each position has an average of one inserted item, leading to almost no `RELABEL` operations during insertion.
- *FEW*: 1 million positions are randomly selected from 10 million items in $\mathcal{O}$, with each position averaging 10 inserted items, possibly triggering a *few* `RELABEL` operations.
- *MANY*: 1,000 positions are randomly selected, with each position averaging 10,000 inserted items, potentially leading to *many* `RELABEL` operations.
- *MAX*: A single position (in the middle of $\mathcal{O}$) is selected to insert 10 million items, triggering the *maximum* number of `RELABEL` operations.

All items are inserted dynamically without preprocessing. This means that 10 million items are randomly assigned to multiple workers, such as 32 workers. Even in the case *MAX*, all insertions are performed sequentially.

A. Evaluating the Running Time

In this experiment, we increased the number of workers from 1 to 64 and measured the actual running time. We perform *INSERT*, *ORDER*, *DELETE*, and *MIXED* in the four test cases: *NO*, *FEW*, *MANY*, and *MAX*.

Fig. 1 illustrates the performance. Two types of locks are compared for performance: the OpenMP lock (dashed lines) and a spin lock implemented with the atomic primitive `CAS` (solid lines). A first glance reveals that the running times generally decrease with an increasing number of workers, except in the *MAX* case for the *INSERT* and *MIXED* experiments. Specifically, we observed the following:

- In the *INSERT*, *DELETE*, and *MIXED* experiments, using the spin lock resulted in significantly faster execution compared to the OpenMP lock. This is because the locked regions have few operations, and busy waiting (Spin Lock) is much quicker than suspension waiting (OpenMP Lock). However, in the *ORDER* experiment, no differences are observed since `ORDER` operations are lock-free and do not require locks for synchronization.
- In the *MAX* case for *INSERT* and *MIXED*, an abnormal trend is noted where the running time increased with the number of workers. This is because the `INSERT` operations in the *MAX* case became sequential since all items are inserted into the same position, leading to the highest contention on shared positions, especially with 64 workers.
- In the *MANY* case for *INSERT* and *MIXED*, the running times decreased until 4 workers are used. However, from 8 workers onward, the running times began to rise. This is due to the `INSERT` operations having only 1,000 positions in the *MANY* case, causing high contention on shared positions when using more than 4 workers.
- In the *ORDER* and *DELETE* experiments, the *MANY* and *MAX* cases consistently performed faster than the *FEW* and *NO* cases. This is because the *FEW* and *NO* cases have 1,000 and 1 operating positions, respectively; these positions are more likely to fit into the CPU cache, and accessing the cache is significantly faster than accessing memory.

VI. Conclusions and Future Work

We show a new parallel Order Maintenance (OM) data structure. The parallel `INSERT` and `DELETE` operations are efficiently synchronized using locks, while the parallel `ORDER` operation is lock-free and can execute with high parallelism.

Our parallel OM data structure has already been applied to the parallel maintenance of the k-order [5], and it can potentially impact other parallel algorithms, such as graph topological order maintenance and parallel ordered set maintenance.

References

[1] P. Dietz and D. Sleator, “Two algorithms for maintaining order in a list,” in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 365–372, 1987.

[2] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito, “Two simplified algorithms for maintaining order in a list,” in *European Symposium on Algorithms*, pp. 152–164, Springer, 2002.

[3] R. Utterback, K. Agrawal, J. T. Fineman, and 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*, pp. 83–94, 2016.

[4] S. Gilbert, J. Fineman, and M. Bender, “Concurrent order maintenance,” *unpublished*, 2003.

[5] B. Guo and E. Sekerinski, “Parallel order-based core maintenance in dynamic graphs,” in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 122–131, 2023.